View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by HAL Mines Nantes

Outillage logiciel pour les problèmes dynamiques

Guillaume Richaud

To cite this version:

Guillaume Richaud. Outillage logiciel pour les problèmes dynamiques. Autre [cs.OH]. Université de Nantes, 2009. Français. <tel-00483061>

HAL Id: tel-00483061

https://tel.archives-ouvertes.fr/tel-00483061

Submitted on 12 May 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITE DE NANTES
FACULTE DES SCIENCES ET DES TECHNIQUES

**ECOLE DOCTORALE SCIENCES ET TECHNOLOGIES
DE L'INFORMATION ET DES MATERIAUX**

Année 2009

N° attribué par la bibliothéque

Outillage logiciel pour les problèmes dynamiques

THÈSE DE DOCTORAT

Discipline : Informatique

Spécialité : Informatique

présentée et soutenue publiquement par

Guillaume Richaud

le 29 octobre 2009 à l'École Nationale Supérieure des Techniques Industrielles et des Mines de Nantes

devant le jury ci-dessous

Président	:	Narendra Jussien, Professeur	École des Mines de Nantes
Rapporteurs	:	Patrice Boizumault, Professeur	Université de Caen
		Christine Solnon, Maître de conférences	Université de Lyon
Examinateurs	:	Gérard Verfaillie, Directeur de recherche	ONERA
		Xavier Lorca, Maître-assistant	École des Mines de Nantes

Directeur de Thèse : Narendra Jussien

ED : .....

OUTILLAGE LOGICIEL POUR LES PROBLÈMES DYNAMIQUES

Outillage logiciel pour les problèmes dynamiques

Guillaume Richaud

Université de Nantes

Sommaire

Chapitre 1

Introduction

1.1 Contexte

En août 2005, British Airways mit quatre jours à rétablir ses vols après une grève d'une journée d'un de ses sous-traitants. En interconnectant et intégrant leurs systèmes d'aide à la décision, les entreprises deviennent de plus en plus soumises aux changements. Parallèlement, avec le développement de technologies comme les puces RFID et la localisation par GPS les entreprises sont capables de suivre en temps réel le déroulement des opérations sur le terrain.

Prenons deux situations concrètes pour illustrer les différentes problématiques (réactivité, nature des perturbations, etc) qui se présente à un utilisateur.

Premier exemple, le cas de la création et de la distribution d'électricité. Les centrales électriques sont connectées entre elles, l'énergie est produite par différents types de générateurs et à des coûts différents. De plus il y a des coûts de démarrage et d'arrêt différents en fonction des générateurs ainsi que des délais différents entre l'allumage et la production effective d'énergie. Un plan de génération et distribution doit aussi satisfaire les limites des différents câbles. Tous les paramètres sont fixés et connus en avance. Les deux facteurs principaux de variation sont les fluctuations des demandes en énergie des différents consommateurs et les pannes imprévisibles. Ces deux sources d'incertitude sont de nature différente et doivent être traiter de manière appropriée. La demande en énergie est une variable qui peut être modélisée précisément en apprenant à partir des données passées. On peut obtenir un plan construit a priori qui prend en compte ces évolutions futures prévisibles et qui ne nécessite que des ajustements mineurs pour s'accorder parfaitement aux fluctuations réelles. Les pannes sont des événements rares et imprévisibles.

Deuxième exemple, le cas d'une pandémie. Lors d'une pandémie les deux seuls moyens d'action sont la vaccination et la quarantaine. Lorsqu'une pandémie est détectée, ce qui est un événement rare, des mesures doivent être prises pour limiter la propagation du virus en utilisant des ressources limitées (doses de vaccins). Une réponse rapide est critique malgré l'incertitude portant sur la vitesse d'infection et l'efficacité du vaccin.

Ces deux exemples illustrent les besoins que peuvent avoir les utilisateurs de problèmes dynamiques. Il peut s'agir de modéliser une variable qui évolue dans un cadre (valeurs extrêmes, instants où surviennent les changements, etc) connu à l'avance, ou de prendre en compte des perturbations survenant de manière imprévisible et qui peuvent remettre en cause le modèle utilisé.

Depuis plusieurs années la programmation par contraintes a prouvé son efficacité pour résoudre de nombreux problèmes : ordonnancement, tournée de véhicules. De plus l'expressivité de la programmation par contraintes permet de modéliser, sous forme de problèmes de satisfaction de contraintes (CSP), des problèmes complexes de manière souple et relativement simple pour l'utilisateur qui n'a pas à se soucier de la façon dont le problème sera résolu. Ces avantages font que la programmation par contraintes est utilisée

dans le domaine industriel. Ainsi, en pratique, les problèmes sont modélisés à l'aide d'un ensemble de variables, qui représentent les différents objets du problème, et d'un ensemble de contraintes, qui expriment les relations liant les variables entre elles. Chaque variable possède un domaine de définition fini (entiers, booléens, etc) correspondant aux différentes valeurs possibles pour la variable. Le but de la programmation par contraintes est de trouver une affectation, pour chaque variable, compatible avec les contraintes. Dans un contexte statique, les contraintes ainsi que les variables et leur domaine sont connus des le début de la résolution. Cependant, comme l'illustrent les exemples précédents, pour être au plus proche des besoins des utilisateurs la programmation par contraintes doit aussi pouvoir prendre en compte l'aspect dynamique des problèmes, c'est a dire, être capable de faire évoluer le modèle pour prendre en compte les événements qui surviennent. On parle alors de problèmes dynamique de satisfaction de contraintes (DCSP). Un DCSP peut être vu comme un ensemble ordonné de CSP, dans lequel le CSP_i varie du CSP_{i+1} par l'ajout/retrait de contraintes. Pour permettre la gestion des DCSP plusieurs méthodes ont été proposées. Le but principal est de permettre la prise en compte des perturbations sans reprendre la résolution depuis zéro.

1.2 Problématique

On peut en tirer des conclusions pour les instances futures en étudiant les instances passées d'un problème. On peut de cette façon construire des solutions robustes aux changements. C'est à dire, des solutions valides (ou qui peuvent s'adapter avec peu de changements) pour un ensemble d'instances variant peu l'une de l'autre. Cependant, lorsque les perturbations sont imprévisibles ou trop nombreuses pour être prises en compte il devient impossible de construire des solutions robustes aux changements qui peuvent survenir. Il s'agit alors de mettre en place des méthodes permettant de réparer et d'adapter le modèle et les solutions aux changements. Au besoin de trouver une nouvelle solution valide s'ajoute le besoin de stabilité dans l'ensemble des solutions, c'est à dire trouver une solution proche (nécessitant peu de changements) de la solution initiale. Cela suppose donc de mettre en place des méthodes de recherche capables de prendre en compte l'aspect dynamique (l'évolution du modèle, l'envie de stabilité, etc) et de l'intégrer à la programmation par contraintes, qui repose habituellement sur un modèle statique fixé au début de la résolution.

1.3 Objectif

L'objectif de cette thèse est de proposer un ensemble d'outils permettant de prendre en compte la dimension dynamique des problèmes. Ainsi en considérant l'aspect dynamique de manière générale (on ne fait aucune sup-

position sur le type de changement qui peut survenir, ni sur quand une perturbation peut survenir) nous proposons plusieurs outils basés sur différentes approches complémentaires pour résoudre les DCSP : l'utilisation des nogoods, la recherche locale, les explication et l'approche sémantique. Notre but est de réduite autant que possible le travail redondant lors de l'adaptation du modèle, tout en permettant une certaine stabilité des solutions.

1.4 Contributions et organisation

La première partie propose une approche des problématiques liés à la dynamicité au travers de problèmes concrets. Le premier chapitre revient sur les notions de stabilité et de robustesse. En utilisant le problème des tournées de véhicules avec fenêtres temporelles comme exemple on introduit les différentes mesures existantes pour évaluer le degré de dynamisme d'un problème. On fait ainsi apparaitre la nécessité de développer de nouvelles mesures pour prendre en compte le point de vue des différents acteurs du problèmes : mesures point de vue client, mesures point de vue entreprises, mesures point de vue livreurs, ... Dans le deuxième chapitre on prend le cas d'un challenge proposé en 2009 portant sur un problème de perturbations pour une compagnie aérienne. Ce challenge nous permet d'illustrer concrètement ce que peut être un problème réel et les difficultés qu'il existe pour trouver une solution efficace sur tous les types de perturbations (panne, grève, problème météorologique, etc) et conciliant les intérêts de tous les acteurs. On présente ensuite le fonctionnement de la programmation par contraintes, les principes sur lesquels elle repose et les différentes méthodes qu'elle utilise pour résoudre un problème dans le cadre statique et une réponse apportée à la question du retrait dynamique de contraintes dans un problème : les explications.

La deuxième partie traite plus précisément des problèmes dynamiques de satisfaction de contraintes. Ainsi dans le premier chapitre on présente un état de l'art des DCSP et des autres différents types de problèmes dynamiques qui peuvent exister. Les différences portent essentiellement sur la façon de prendre en compte le caractère dynamique du problème (l'intégrer au modèle, aux méthodes de résolution). On présente aussi les différentes méthodes de résolution associées. Dans le deuxième chapitre on présente une méthode de recherche locale utilisée avec succès dans le cadre statique dont la spécificité est de reposer sur la programmation par contraintes pour construire le voisinage d'une solution. On propose une nouvelle méthode, pour calculer le voisinage d'une solution, en utilisant les explications. Dans le deuxième chapitre, on s'intéresse aux nogoods. Leur utilisation permet d'enregistrer les parties de l'espace de recherche qui ont été explorées sans succès. Les nogoods sont donc utilisées pour limiter le travail redondant lorsqu'on a à résoudre un problème proche d'un problème qu'on a déjà résolu précédemment. On propose une contrainte basée sur un automate pour stocker de manière compacte et dyna-

miquement les ensembles de nogoods découverts pendant la recherche. Dans le troisième chapitre, on se concentre sur le cas des contraintes globales. On part de la constatation que les propriétés des contraintes peuvent être exploitées avantageusement pour limiter le travail redondant. On étend donc des travaux préexistants sur la monotonie des contraintes au cas du retrait de variable et on présente la représentation des contraintes sous formes de graphes et de propriétés de graphe pour en déduire les propriétés des contraintes. On se concentre ensuite sur la question de l'incrémentalité dans les contraintes en proposant une illustration au travers de la contrainte `tree`. Nos premières questions sur l'incrémentalité nous mènent ensuite à nous intéresser à la relation contrainte-solveur et à comment développer des contraintes incrémentales indépendantes d'un solveur. L'incrémentalité et l'indépendance par rapport au solveur sont des éléments importants dans le cadre dynamique puisque c'est grâce à eux qu'on peut adapter l'instanciation d'un modèle durant la résolution sans avoir à reprendre depuis zéro la recherche en cours.

Nos différentes contributions sont donc :

1. le développement de mesures de proximité, prenant en compte le point de vue des différents acteurs, entre solutions,
2. le développement d'une contrainte basée sur les automates permettant l'ajout et le retrait dynamique de tuples durant la recherche,
3. l'utilisation des explications pour construire le voisinage d'une solution dans le cadre d'une recherche locale,
4. le développement d'une version incrémentale et portable de la contrainte `tree` et la généralisation possible de la portabilité aux contraintes globales.

Première partie

Les problèmes dynamiques

Chapitre 2

Problèmes dynamiques et modélisation

2.1 Un problème concret : gestion des perturbations pour une compagnie aérienne

Depuis 1999 la Société française de Recherche Opérationnelle et Aide à la Décision (ROADEF) lance un challenge dédié aux applications industrielles. Il permet aux industriels d'avoir une meilleure perception des développements récents dans le domaine de la recherche opérationnelle et de l'aide à la décision. Il confronte de jeunes chercheurs à une problématique décisionnelle, souvent complexe, rencontrée dans le milieu industriel. Après l'ONERA, le CNES, Renault et France Telecom, l'entreprise Amadeus a proposé en 2009 un challenge portant sur la gestion des perturbations dans le domaine aérien. Il s'agissait d'adapter la planification des vols aux perturbations afin de minimiser l'impact pour les voyageurs et sur la planification prévue.

Au travers de cet exemple concret nous allons illustrer, les questions et les difficultés que peuvent rencontrer des utilisateurs confrontés à des problèmes dynamiques. Cet exemple permettra aussi d'aborder les différentes dimensions d'un problème dynamique : nature des perturbations, impacte des utilisateurs, nécessité de retourner à la planification connue, etc.

2.2 Définition du problème

Le but d'une compagnie aérienne est d'acheminer le plus de passagers possibles. Elle dispose pour cela d'un ensemble d'avions effectuant différents vols entre les aéroports. Lors d'une perturbation, les passagers voyageant sur des vols perturbés doivent adapter leur itinéraire pour atteindre leur destination (avec les désagréments que cela peut induire). Afin d'introduire le problème nous allons présenter les différents acteurs à prendre en compte lors de la recherche d'une planification aérienne. Nous présenterons ensuite les différents types de perturbations et les couts des désagréments pour les passagers.

2.2.1 Les différents éléments du problème

Le problème fait intervenir différents acteurs : les aéroports, les vols, les avions et les voyageurs. Chacun possède des caractéristiques et doit respecter des contraintes qui lui sont propres.

Les aéroports Un aéroport est caractérisé par un nombre de décollages (atterrissages) autorisés pour chaque tranche horaire et par sa distance par rapport à chacun des autres aéroports. La distance entre deux aéroports est ramenée à la durée d'un vol allant de l'un à l'autre. Les distances ne sont pas forcément symétriques. Une solution compatible pour un aéroport est un ensemble de vols ne dépassant pas sa capacité d'atterrissage et de décollage.

Les vols Un vol est caractérisé par un aéroport et une heure de départ, ainsi que par un aéroport et une heure d'arrivée. La durée d'un vol dépend uniquement de la distance horaires entre les deux aéroports reliés (indépendante du type d'appareil utilisé).

Les voyageurs Un voyageur est caractérisé par un aéroport de départ et un aéroport d'arrivée. Une solution, pour un voyageur, est un itinéraire c'est à dire une séquence de vols permettant de relier l'aéroport de départ à l'aéroport d'arrivée en respectant la continuité du voyage : l'aéroport d'origine d'un vol correspond à l'aéroport de destination du vol précédent et sa date de départ est postérieure à la date d'arrivée de ce dernier augmentée du temps de transit (temps minimal nécessaire pour une correspondance).

Les avions Un avion est caractérisé par son modèle, temps de réengagement (temps minimal necéssaire pour préparer l'avion pour un nouveau vol et permettre le désembarquement/embarquemant des passagers, le nettoyage de l'appareil et le changement d'équipage), temps de transit (temps minimal entre deux vols ne necessitant ni de changement d'équipage, ni de nettoyage de l'appareil ou de désembarquement des passagers), distance maximale franchissable, ensemble des configurations possibles ,sa configuration fixe (répartition des sièges dans chaque cabine) ainsi que ça durée de vol avant révision et son aéroport de départ. Pour un avion, une solution est une rotation : c'est à dire une séquence de vols successifs qu'il doit réaliser et qui vérifient la continuité des opérations : l'aéroport d'origine d'un vol correspond à l'aéroport de destination du vol précédent et sa date de départ est postérieure à la date d'arrivée de ce dernier augmentée du temps de réengagement (temps minimal nécessaire pour préparer l'appareil pour un nouveau vol). De plus, l'avion ne doit pas dépasser son nombre d'heures de vols avant révision et respecter les contraintes liées à son modèle (distance maximale franchissable).

Une solution initiale, satisfaisant ces contraintes, est donnée. Le but du challenge est de réadapter cette solution pour prendre en compte les perturbations qui ont pu survenir.

2.2.2 Les perturbations

Le but du problème est de rétablir l'ordre normal des opérations, en cas de perturbation ayant une incidence sur le programme de vols défini, dans les meilleurs délais possibles. Il s'agit donc de générer un nouveau programme de vols pour la période de recouvrement définie. Autrement dit, on détermine, pour chaque appareil, un rotation prenant en compte les perturbations. Le programme de vols proposé doit minimiser les coûts induits et les impacts sur les passagers.

La nature des perturbations varie selon les incidents rencontrés :

- **retard de vol**, les causes peuvent être un temps d'embarquement allongé, une grève des personnels au sol, un temps de préparation de l'appareil augmenté, l'attente d'un équipage et d'un passager en correspondance ;
- **annulation de vol** causée par des problèmes d'équipage ;
- **indisponibilité d'un appareil** pour une période donnée pour cause de panne ou de problemes techniques. Aucun vol ne peut lui être affecté pendant toute la durée de son indisponibilité ;
- **réduction du nombre de vols opérés**, au niveau d'un aéroport, sur un intervalle de temps donné. Ce cas de figure apparait lors de conditions météorologiques défavorables (réduction du nombre de départs/arrivées aux aéroports) ou lors d'une grève du personnel (réduction du nombre de vols pour cause de sous-effectif)

Le nouveau programme de vols résulte de la prise d'un certain nombre de décisions portant sur les vols du programme initial : annulations et retards intentionnels, changements d'appareil au sein de la même famille et créations éventuelle de nouveaux vols.

Les critères de décisions concernent essentiellement la minimisation des coûts pour la compagnie, aussi les modifications du programme de vols peuvent elles être effectuées aux dépens des passagers. Afin de pallier cette lacune, un critère de minimisation des perturbations subies par les passagers est intégré à la prise de décision.

Les passagers sur les vols impactés (vols du programme initial qui subissent un changement) rencontrent potentiellement des perturbations qui nécessitent de trouver de nouveaux itinéraires ou d'annuler leur voyage. Ces modifications d'itinéraire sont évaluées, non seulement en terme de coûts de retard ou d'annulation, mais aussi de confort.

2.2.3 Coûts d'une solution pour les passagers

Suite aux perturbations, les passagers peuvent ressentir différents désagréments : retards, voyage dans des cabines de classe inférieure, annulation du voyage. Ainsi un cout est associé a chaque type de désagréments afin d'évaluer l'impact des acheminements sur les passagers. Dans cette section on définit la notation et les différents critères pris en compte pour calculer un passager.

2.2.3.1 Notations

Soit $\mathcal{V}$ un ensemble de vols où chaque vol v est défini par un quadruplet (s, d, t_s, a) avec $s \in \mathcal{A}$ l'aéroport de départ, $d \in \mathcal{A}$ l'aéroport d'arrivée, t_s la date de départ et $a \in \mathcal{K}$ l'avion associé à ce vol. Ainsi à chaque vol on peut associer une capacité $cap = cap_a$, une durée de vol $t_t = t_{(s,d)}$ et une date d'arrivée $t_d = t_t + t_s$.

Soit $\mathcal{P}$ un ensemble de groupe de passagers où chaque groupe p de passagers est défini par un tuple (x, s, d, i, r) avec x le nombre de passagers concernés, $s \in \mathcal{A}$ l'aéroport de départ, $d \in \mathcal{A}$ l'aéroport d'arrivée, i l'itinéraire initial et $r \in \{aller, retour\}$ permettant de différencier les itinéraires retour des itinéraires aller. Soit $[v_0, \ldots, v_k]$ la séquence de vols composant l'itinéraire i. On peut donc associer à chaque groupe de passagers une date de départ $t_s = t^{v_0}$ et une date d'arrivée $t_d = t_a^{v_k}$.

Étant donné un ensemble de vols $\mathcal{V}$, on cherche des itinéraires pour chaque groupe de passagers minimisant le coût global pour la compagnie. Un itinéraire valide pour un groupe de passagers g est constitué d'une séquence de vols $[v_0, \ldots, v_i, v_{i+1}, \ldots, v_k]$ telle que :

- $s^{v_0} = s^g$ (aéroport d'arrivée)
- $d^{v_k} = d^g$ (aéroport de départ)
- $\forall i \in [0 \ldots k-1], d^{v_i} = s^{v_{i+1}}$ (continuité spatiale)
- $\forall i \in [0 \ldots k-1], t_d^{v_i} + t_{transit} = t_s^{v_{i+1}}$ (continuité temporelle)

2.2.3.2 Définition des coûts

À chaque itinéraire est associé un coût dépendant du retard final et des déclassements subis par un passager l'empruntant. Ainsi pour un groupe de passagers g :

$$c^g(i,t) = \alpha \left(C^g_{delay-legal} + C^g_{cancel-legal} \right) + \beta \left(C^g_{down} + C^g_{delay-pax} + C^g_{cancel-pax} \right)$$

Avec :

- α et β deux coefficients fixés lors de la définition du problème,
- $C^g_{delay-legal}$ le coût de retard d'un passager du groupe g (fonction de la durée du voyage et du retard),
- $C^g_{cancel-legal}$ le coût d'annulation du voyage d'un passager du groupe g (prix du billet et compensation additionnelle dépendant de la durée du voyage),
- C^g_{down} le coût de déclassement d'un passager du groupe g dépendant du vol et de la cabine de référence,
- $C^g_{delay-pax}$ le coût de retard d'un passager du groupe g (type de l'itinéraire, cabine de référence, retard constaté, trajet aller/retour),
- $C^g_{cancel-pax}$ le coût d'annulation du voyage d'un passager du groupe g (type de l'itinéraire, classe à laquelle le passager voyage, retard constaté, trajet aller/retour).

Ces différents coûts définissent l'impact des déclassements/retard/annulation du point de vue d'un passager. En pratique il s'agit des pénalites les plus importantes pour la compagnie. Le coût le plus important pour la compagnie est le cout d'annulation du voyage pour un

passager. Ainsi, lors de la construction d'une solution, il est important de minimiser le nombre de passagers dont le voyage doit être annulé.

2.3 Méthode de résolution

Trouver une bonne solution signifie, d'une part, trouver un nouveau plan de vols pour les avions (en respectant les contraintes sur les avions et les aeorports) et, d'autre part, trouver un nouvel itinéraire pour chaque passager. Cela implique que le nouveau plan de vol permettent l'acheminement d'un maximum de passagers. La difficulté du challenge repose sur la cohabitation de ces deux objectifs liés au travers des couts de pénalités.

On considère la génération des itinéraires des passagers comme un sous-problème. Le problème maître consiste alors à générer les rotations des avions en prenant en compte l'intérêt que représente chacun des vols pour les passagers. Notre but est donc de définir un nouveau programme de vols minimisant les couts liés aux passagers.

Nous décrivons ici la décomposition et le schéma général de résolution de la méthode réalisée (section 2.3.2), le détail des algorithmes mis en œuvre dans la résolution des sous-problèmes (section 2.3.3) et les améliorations possibles (section 2.4.1). La section 2.3.1 présente les notations et définitions des notions introduites par la suite.

2.3.1 Définitions et notations

Soit A l'ensemble des aéroports et T la séquence des périodes de 5 minutes incluses dans la période de planification : $[T_0, T_0+5[, [T_0+5, T_0+10[, \ldots, [T_1-5, T_1[$. Soit le graphe orienté $G = (V, E)$ sur l'ensemble des sommets $V = A \times T$ et l'ensemble des arcs $E = \{((a_1, t_1), (a_2, t_2)) \in V \times V \ | t_2 \geq t_1\}$.

La *rotation* r associée à un avion peut être modélisée comme un chemin dans G, vide ou qui contient en alternance des arcs $((a_1, t_1), (a_2, t_2))$ et $((a_2, t_2), (a_3, t_3)) \in E$ tels que $a_1 \neq a_2$, $a_2 = a_3$ et $t_2 + t_{reop} \leq t_3$. Les arcs du premier type sont appelés des *vols*, ceux du second type, des *correspondances*.

Le plan de vol d'une flotte d'avions est un ensemble de chemins disjoints dans G, et forme donc un sous-graphe de $G_R = (V, E_R)$ de G (Figure 2.1).

L'*itinéraire* i associé à un groupe de passagers peut être modélisé comme un chemin dans G_R, vide ou qui contient en alternance des arcs $((a_1, t_1), (a_2, t_2))$ et $((a_2, t_2), (a_3, t_3)) \in E_R$ tels que $a_1 \neq a_2$, $a_2 = a_3$ et $t_2 + t_{correspondance} \leq t_3$.

L'ensemble des itinéraires des passagers représente un ensemble de chemin dans G_R tel que pour chaque arc $((a_1, t_1), (a_2, t_2)) \in E_R$ de type *vol*, la capacité de l'avion réalisant ce vol est supérieur au nombre de passagers qui lui sont affectés.

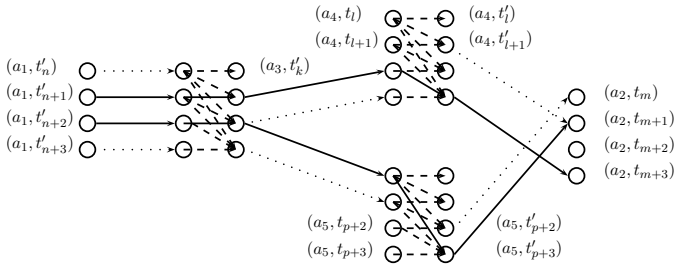

FIGURE 2.1: L'ensemble des noeuds et des arcs forme le graphe $G = (V, E)$. Deux rotations sont représentées : la rotation composée par les arcs $((a_1, t'_{n+1}), (a_3, t_{k+1}))$, $((a_3, t_{k+1}), (a_3, t'_{k+1}))$, $((a_3, t'_{k+1}), (a_4, t_{l+2}))$, $((a_4, t_{l+2}), (a_4, t'_{l+3})$, $((a_4, t'_{l+3}), (a_2, t_{m+3}))$ et la rotation constitué de $((a_1, t'_{n+2}), (a_3, t_{k+2}))$, $((a_3, t_{k+2}), (a_3, t'_{k+2}))$, $((a_3, t'_{k+2}), (a_5, t_p))$, $((a_5, t_p), (a_5, t'_{p+3})$ et $((a_5, t'_{p+3}), (a_2, t_{m+3}))$.

2.3.2 Schéma général de décomposition

La méthode de résolution proposée suit la décomposition intuitive du problème :

1. conception d'un nouveau plan de vol de la flotte (réparation des rotations affectées par les perturbations)
2. conception des itinéraires passagers sur ces rotations

À chaque itération on construit un nouveau plan de vol qui respecte l'ensemble des contraintes dures du problèmes (temps minimum entre deux vols, distance maximale franchissable, temps de vol avant révision de l'avion, nombre d'atterrissages et de décollages autorisés pour chaque aéroport). De ce plan de vol on déduit un ensemble de vols disponibles (ainsi que le nombre de places disponibles sur ce vol) et utilisables par les passagers. On cherche ensuite pour chaque groupe de passagers l'itinéraire de coût minimal. L'objectif est d'utiliser ensuite les causes des différents échecs rencontrés lors de la génération des itinéraires pour orienter la réparation des rotations à l'itération suivante.

2.3.3 Résolution des sous-problèmes

2.3.3.1 Prétraitement des perturbations

Afin de trouver rapidement un ensemble de rotations répondant aux perturbations, nous commençons par traduire les réductions de capacités des aéroports et les indisponibilités des avions en annulation de vols. Ainsi si le nombre d'atterrissages et de décollages autorisé pour un aéroport devient insuffisant pour permettre les vols initialement prévus nous créons de nouvelles perturbations annulant les vols les moins intéressants a priori. De même si un

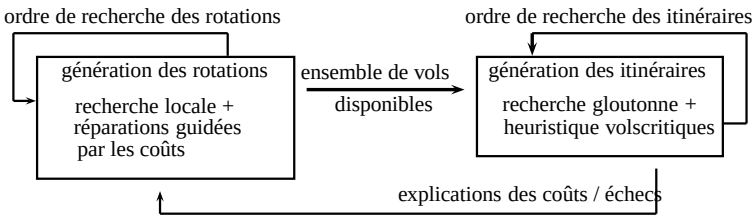

FIGURE 2.2: Le problème est décomposé en deux sous-problème (2.3.3) . Le SP1 (2.3.3.2) correspondant à la réparation des rotations et le SP2 (2.3.3.3) correspondant à la recherche des itinéraires pour chaque groupe de passagers. Les deux sous-problèmes interagissent via les vols (coûts, explications).

avion vient à être immobilisé nous créons de nouvelles perturbation annulant les vols que l'avion devait assurer durant cette période. L'avantage est que cela nous permet d'utiliser nos méthodes de réparation pour les annulations de vols / retard de vols pour prendre en compte les perturbations d'aéroports et d'avions.

2.3.3.2 Réparation des rotations

Actuellement les modifications réalisées sur les rotations pour prendre en compte les perturbations sont des réparations locales afin d'assurer la continuité de la rotation initiale. Ainsi lorsqu'un vol est retardé la réparation consiste à retarder les vols suivants dans la rotation. Lorsqu'un vol entre un aéroport $airport_{source}$ et $airport_{dest}$ est annulé, la réparation va consister en la création d'un nouveau vol reliant $airport_{source}$ à un aéroport par lequel l'avion était sensé passer dans la suite de sa rotation (figure 2.3). Cette réparation nous permet de rester proche de la rotation initialement prévue.

La réparation liée à l'annulation d'un vol pour cause de problème de capacité d'un aéroport reprend le même principe sauf qu'au lieu de créer un vol direct entre l'aéroport source et un des aéroports suivants, on tente de créer un ensemble de vols passant par des aéroports fortement connectés avec les aéroports devenus inaccessibles (figure 2.3). L'objectif est que les passagers puissent utiliser des correspondances pour rallier leurs destinations.

La limitation des capacités des aéroports fait que les avions dont on répare la rotation en premier ont un plus large choix d'aéroports par lequel il peut passer. L'ordre dans lequel on traite les avions est donc important et nous essayons de traiter en premier ceux qui semblent le plus critiques (nombre de passager, type de liaison assurée).

2.3.3.3 Recherche des itinéraires

Une fois un ensemble de rotations déterminé, on dispose alors d'un ensemble de vols. Parmi ces vols, on cherche l'itinéraire le moins coûteux pour

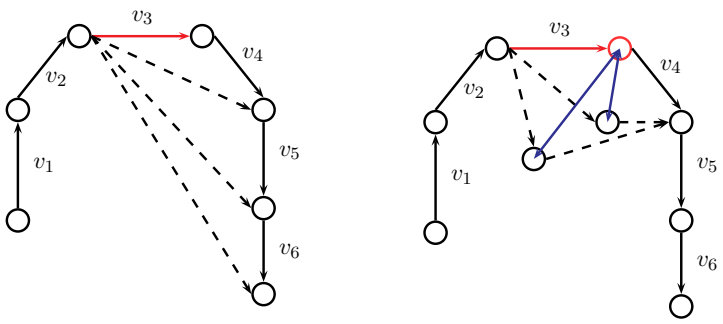

FIGURE 2.3: Le vol v_3 est pérturbé. Réparation d'une rotation lors de l'annulation d'un vol (à gauche) et lorsqu'un vol devient irréalisable à cause de la limite de capacité d'un aéroport (à droite). Les arcs en pointillés correspondent aux vols créés pour essayer de réparer la rotation

chaque groupe de passager. Pour trouver le chemin de coût minimum pour un groupe de passager on utilise l'algorithme du Max Flow - Min Cost [AMO93] sur un graphe de flots (figure 2.4) construit à partir des vols disponibles (figure 2.3.3.3). Le but de cette représentation est de faire figurer dans une même représentation, les capacités des vols, les coûts et les correspondances possibles (figure 2.6).

Pour chaque vol v on crée deux nœuds v_{source} et v_{dest} reliés par un arc dont la capacité est égale au nombre de places disponibles dans l'avion assurant ce vol et dont le coût dépend du retard et du déclassement que cela engendre par rapport à l'itinéraire initial du passager. On ajoute aussi les arcs (v_{dest}, w_{source}) où w est un vol qu'un passager peut prendre après le vol v et (u_{dest}, v_{source}) où u est un vol qu'un passager peut prendre avant le vol v. Ces arcs "correspondances" n'ont pas de limite de capacité et ont un coût dépendant du retard que l'attente entre les deux vols engendre. Afin de prendre en compte la possibilité d'annulation du voyage pour un groupe de passager on ajoute en plus un arc reliant l'aéroport de départ à l'aéroport d'arrivée et dont le coût est égal au coût d'annulation du voyage.

La limite de cette méthode est que l'ordre dans lequel on considère chaque groupe de passagers est important. En effet les passagers pour qui on cherche un itinéraire en premier disposent d'un plus grand nombre de vols sur lesquels il reste de la place. On peut ainsi affecter des passagers à un vol qui deviendra critique par la suite alors qu'il existe un autre itinéraire pour ces passagers. Pour être plus efficace nous avons donc commencé à mettre en place quelques stratégies sur l'ordre dans lequel doivent être traités les passagers.

Résolution Un graphe $(\mathcal{A}, \mathcal{V})$ ne permet pas de représenter les précédences entre les différents vols composant un itinéraire (cf Fig. 2.3.3.3). Ainsi, pour la recherche des itinéraires on travaillera à partir d'un graphe $(\mathcal{V}, \mathcal{C})$, où $\mathcal{C}$

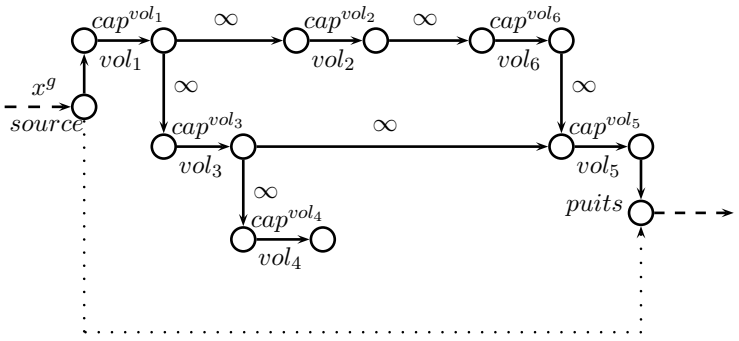

FIGURE 2.4: Graphe de flot construit à partir de l'ensemble des vols disponibles et représentant les correspondances entres les différents vols.

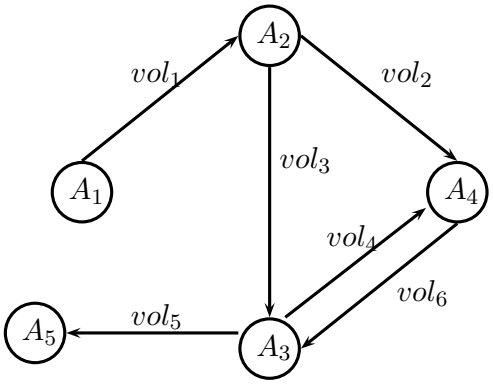

FIGURE 2.5: Graphe des vols disponibles après la réparation des rotations.

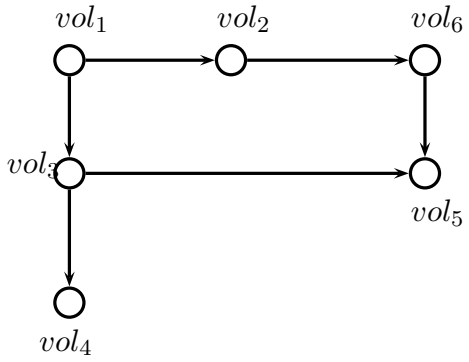

FIGURE 2.6: Graphe des correspondances possibles entre les vols.

correspond à l'ensemble des correspondances possibles entre deux vols (Fig. 2.6). Une correspondance entre un vol u et un vol v est possible si $d^u = s^v$ et $t_d^u + t_{correspondance} \leq t_s^v$.

On transforme ensuite $\mathcal{G}_{cor}$ en $\mathcal{G}_{cap} = (\mathcal{V}_s \cup \mathcal{V}_d, \mathcal{C} \cup \mathcal{V})$ afin de prendre en compte les capacités (cf Fig. 2.4). À chaque arc de $\mathcal{C}$ on associe une capacité ∞ et à chaque arc de $\mathcal{V}$ on associe la capacité de l'avion assurant le vol.

On cherche une répartition de coût minimal des passagers sur les différents itinéraires possibles. Pour chaque groupe de passagers g, on cherche donc à résoudre un problème de Maximum Flow Minimum Cost [1] [AMO93] dans $\mathcal{G}_{cap}^g$, c'est à dire dans le graphe $\mathcal{G}_{cap}$ au quel on a ajouté un noeud *source* relié aux vols décollant depuis s^g, un noeud *puits* au quel sont reliés les nœuds atterrissant à d^g, un arc allant de *source* à *puits* de coût $\alpha C_{cancel-legal}^g + \beta C_{cancel-pax}^g$ et un arc allant de *puits* à *source* de capacité x^g. De plus, on associe un coût de C_{down}^g à chacun des arcs de $\mathcal{V}$ ainsi qu'un surcout $\alpha C_{delay-legal}^g i + \beta C_{delay-pax}^g i$ dépendant de leur retard par rapport à la date d'arrivée initialement prévue.

Afin de réduire la taille du graphe et la complexité des algorithmes utilisés on effectue plusieurs simplifications. On peut relaxer le problème en ne prenant plus en compte le déclassement dans le calcul des coûts. Les capacités des avions se limitent alors aux places disponibles dans l'avion (indépendamment de la classe de la cabine à la quelle appartiennent les places). Une autre simplification peut être de ne prendre en compte que le cout d'annulation d'un voyage (puisqu'il est très supérieur aux autres couts). Les arcs du graphe ont alors un coût nul (hormis l'arc correspondant a l'annulation du voyage). Afin de réduire le nombre d'arcs "correspondances" (potentiellement un passager peut prendre n'importe quel vol, respectant les contraintes horaires, au départ d'un aéroport dans lequel il est arrivé) dans la représentation graphe on crée des arcs entre les différents vols au départ du même aéroport. Les arcs vont dans le sens chronologique des heures de décollage des différents vols. Ainsi au lieu de faire apparaitre tous les arcs de correspondances possibles, on ne prends en compte que l'arc de correspondance au plus tôt (figure 2.3.3.3).

1. Le problème du Maximum Flow Minimum Cost combine un problème de Maximum Flow avec un problème de Minimum Cost. Il s'agit de trouver le flot maximum reliant un nœud source s à un nœud destination t tel que le coût du flot soit le coût minimum. Autrement dit on cherche un flot de coup minimal parmi les flots de capacité maximale.

instance	UB	CPU
A01_6088570	158485, 35	600
A02_6088570	522458, 40	600
A03_6088570	578302, 90	600
A04_6088570	6389528, 10	600
A05_6088570	42546371, 90	600
A06_6088590	125286, 85	600
A07_6088590	592338, 40	600
A08_6088590	1308968, 05	600
A09_6088590	9373957, 95	600
A10_6088590	59911082, 45	600

FIGURE 2.8: Résultats obtenus pour les 10 instances tests de la base A.

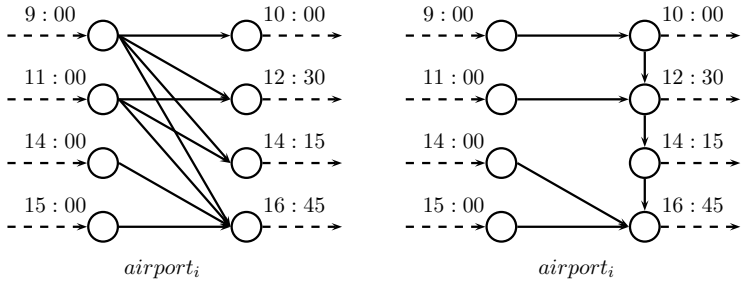

FIGURE 2.7: Réduction du nombre d'arcs correspondance

2.4 Résultats expérimentaux – Base A

Le tableau ci-dessous (tableau 2.8) récapitule les résultats obtenus sur les 10 instances tests de la base A (SA1_6088570 et SA2_6088590) fournit par les organisateurs du challenge. Une instance se présente sous la forme d'un plan de vols, d'un ensemble d'itinéraires auxquels sont initialement affectés les passagers et d'un ensemble de perturbations. On connait de plus les informations relatives aux aéroports, aux avions et aux coûts de pénalité.

Le tableau comporte trois colonnes :

1. colonne `instance` : nom de l'instance
2. colonne `UB` : coût de la solution retournée
3. colonne `CPU` : temps d'exécution du programme en secondes

Les autres tableaux reprennent les résultats des différentes approches proposées. La majorité des équipes ont séparé les deux aspects du problème : la planifications des vols des avions et les itinéraires des passagers. Ainsi la première équipe a traité le problème sous la forme de deux problèmes de flots

Equipe	*A*01	*A*02	*A*03
Bisaillon, Cordeau, Laporte, Pasin	29891,75	**116431,70**	202358,10
Hanafi, Wilbaut, Mansi	**28155,60**	126684,70	**131694,60**
Acuna-Agost, Michelon, Feillet, Gueye	44938,40	374763,75	672790,35
Darlay, Kronek, Schrenk, Zaourar	317525,05	765351,40	438681,05
Jozefowiez, Mancel, Mora-Camino	150095,70	377992,90	473992,15
Eggermont, Firat, Hurkens, Modelski	987236,15	770971,35	1588231,40
Demassey, Jussien, Lorca, Menana, Richaud	158485,35	540895,65	567124,65
Dickson, Smith, Li	1345066,80	986506,60	1794442,25
Estellon, Gardi, Nouioua	297019,05	928384,65	2180182,40
Eggenberg, Salani	3860511,50	721588,60	2395934,10

FIGURE 2.9: Résultats obtenus pour les différentes approches proposées. En gras le plus petit coût trouvé. Les équipes sont classées de manière décroissante. Les premières correspondent aux équipes ayant renvoyé des résultats moyens de coûts inférieurs.

multi-resources. Le premier porte sur les avions passant par les aéroports et le second sur les passagers voyageant via les vols. Les deux sont modélisés en utilisant une formulation Mixed Integer Programming [WN99]. L'efficacité de leur approche tient en grande partie a la stratégie (Statical Analysis of Propagation of Incidents) qu'ils ont mis en place pour réduire l'espace de recherche (fixer certaines variables). Elle part de la constatation que tout les éléments de la solution initiale n'ont pas la même probabilité d'être remis en cause par une perturbation donnée. Ainsi durant la résolution ils attachent a chaque vol un probabilité qu'il soit remis en cause dans une solution optimale et en déduisent les vols à fixer.

De la même façon, la deuxième équipe utilise un modélisation basée sur MIP pour dans un premier temps résoudre le problème des itinéraires des avions et de s'assurer que les contraintes qui y sont liées sont respectées, puis dans un deuxième temps s'occuper d'affecter les passagers aux différents vols. Ils tentent ensuite d'améliorer la solution obtenues en explorant son voisinage (retrait/ajout des routes, d'itinéraires, etc).

La troisième équipe se concentre sur une méthode de réparation qui tente de remplacer, ce qui a été supprimé par les perturbations, en créant des vols similaires. Dans la première phase ils se concentrent donc sur la réparation du plan de vol et tentent ensuite de réaffecter les passagers aux nouveaux vols crées.

Il y a 10 instances, dont la difficulté varie essentiellement à cause du type de perturbations survenant. Les instances *A*01, *A*02, *A*06 et *A*07 sont perturbées par des retards dans certains vols, aux quels s'ajoutent l'immobilisation d'un appareil dans les instances *A*03 et *A*08. Les instances *A*04 et *A*09 sont perturbées par une baisse ponctuelle de capacité de quelques aéroports

Equipe	*A*04	*A*05	*A*06
Bisaillon, Cordeau, Laporte, Pasin	139747,10	**3717376,35**	**44305,05**
Hanafi, Wilbaut, Mansi	523678,80	8092977,55	71355,25
Acuna-Agost, Michelon, Feillet, Gueye	**108081,90**	16134976,15	86283,00
Darlay, Kronek, Schrenk, Zaourar	789399,05	8012044,00	269462,00
Jozefowiez, Mancel, Mora-Camino	2520586,00	13640667,40	111540,50
Eggermont, Firat, Hurkens, Modelski	1573294,50	23990055,70	1032790,50
Demassey, Jussien, Lorca, Menana, Richaud	4842154,70	41037750,05	125286,85
Dickson, Smith, Li	2406499,25	36011581,05	1389306,45
Estellon, Gardi, Nouioua	1733066,60	42259652,80	247315,35
Eggenberg, Salani	6161533,20	43978009,85	4980974,40

FIGURE 2.10: Résultats obtenus pour les différentes approches proposées. En gras le plus petit coût trouvé.

Equipe	*A*07	*A*08	*A*09
Bisaillon, Cordeau, Laporte, Pasin	**202247,75**	659572,00	215482,35
Hanafi, Wilbaut, Mansi	245696,70	**329521,85**	1096303,85
Acuna-Agost, Michelon, Feillet, Gueye	452752,85	1065896,20	**187144,40**
Darlay, Kronek, Schrenk, Zaourar	546110,00	799238,00	938095,80
Jozefowiez, Mancel, Mora-Camino	623236,80	997137,80	6163295,90
Eggermont, Firat, Hurkens, Modelski	834044,65	1559644,90	1478106,60
Demassey, Jussien, Lorca, Menana, Richaud	634489,95	1320747,05	8750676,50
Dickson, Smith, Li	883862,40	1949700,65	2529246,70
Estellon, Gardi, Nouioua	1031022,70	2586730,75	2205772,30
Eggenberg, Salani	1545630,05	5048257,00	10509766,30

FIGURE 2.11: Résultats obtenus pour les différentes approches proposées. En gras le plus petit coût trouvé.

et quelques retards contrairement aux instances *A*05 et *A*10 pour les quelles les aéroports sont frappés par de nombreuses réductions de capacités de très longue durée.

On observe une certaine stabilité des performances des différents approches en fonction du type de perturbations rencontrée par l'instance. Ainsi la solution proposée par la première équipe semble particulièrement adaptée aux instances frappées par un retard de vols, celle de l'équipe 2 au cas des pannes d'appareils et celle de l'équipe 3 au cas des baisses ponctuelles de capacité. Malgré des approches assez similaires entre les différents concurrent, on note la difficulté qu'il y a a développer une solution permettant de gérer efficacement tout les cas de figure.

Pour être efficace une approche doit donc faire coopérer, intelligemment, plusieurs méthodes complémentaires et permettant de traiter de manière

Equipe	*A*10
Bisaillon, Cordeau, Laporte, Pasin	**7210166,90**
Hanafi, Wilbaut, Mansi	17166321,75
Acuna-Agost, Michelon, Feillet, Gueye	20892659,25
Darlay, Kronek, Schrenk, Zaourar	15098771,00
Jozefowiez, Mancel, Mora-Camino	23840247,85
Eggermont, Firat, Hurkens, Modelski	29639201,15
Demassey, Jussien, Lorca, Menana-Quillet, Richaud	58046489,60
Dickson, Smith, Li	44798632,70
Estellon, Gardi, Nouioua	58333715,85
Eggenberg, Salani	69097236,55

FIGURE 2.12: Résultats obtenus pour les différentes approches proposées. En gras le plus petit coût trouvé.

spécifique les différentes sous parties du problème. En effet les problèmes réels cherchent souvent à répondre à plusieurs objectifs et a saisir simultanément plusieurs dimension d'un problème. Ainsi, dans ce challenge, les participants devaient prendre en compte les contraintes portant sur les vols, la partie réparation liée aux perturbations et la partie coût liée a la réaffectation des passagers.

2.4.1 Perspectives d'amélioration

Un aspect de notre méthode à améliorer est la prise en compte des informations "itinéraires" pour guider la réparation des rotations des avions. Actuellement la communication entre les deux sous-problèmes se fait essentiellement de la planification des rotations vers le calcul des itinéraires (via l'existence ou non des vols). Mais des informations comme : les raisons (vols annulés, vols trop retardés) pour lesquelles un itinéraire a été annulé pourraient être utilisés pour privilégier certains vols ou pour privilégier certaines rotations. Notre objectif est donc d'intégrer des informations plus précises sous la forme d'explications afin de guider la phase de réparation des rotations.

2.5 Conclusion

Nous venons de voir qu'un problème fait souvent intervenir plusieurs dimensions et préoccupations. Il en est de même pour les problèmes dynamiques. En fonction des préoccupations des différents acteurs un aspect et une approche peut être privilégié à une autre. La difficulté tient au fait qu'il faut prendre en compte des intérêts contraires (satisfaction des utilisateurs, gains, etc). Il faut ainsi réussir a trouver des approches permettant de prendre en compte les différentes attentes sans en privilégier une.

Au travers de l'exemple du challenge ROADEF est apparu la nécessité de pouvoir combiner plusieurs types d'approches pour résoudre un problème réel. La question du choix de la technologie à utiliser se pose aussi lorsqu'il s'agit de traiter un problème dynamique. Nous avons décidé de nous intéresser aux problèmes dynamiques pour les quels on ne possédé aucune information sur les évolutions futures et toutes les transformations (aussi bien sur les données que sur le modèle) peuvent survenir. Nous avons décidé d'utiliser la programmation par contraintes pour traiter ce problème en raison du caractère déclaratif qui permet une modélisation haut niveau, ainsi qu'en raison de l'utilité que représente la programmation par contraintes pour la résolution des problèmes réels.

Chapitre 3

Problèmes dynamiques et évaluation de la stabilité

3.1 Présentation des problèmes dynamiques : Motivations et intérêt du sujet

Avec le besoin croissant de pouvoir s'adapter en temps-réel aux variations qui ont lieu sur le terrain, les problèmes dynamiques jouent un rôle de plus en plus importants. Contrairement aux problèmes statiques pour lesquels on cherche une solution, les problèmes dynamiques consistent à transformer une solution en cours d'utilisation en une solution pour un problème légèrement différent. Des mesures ont déjà été proposées pour quantifier le degré de dynamisme d'un problème. Ces différentes mesures se basent sur les perturbations du modèle (nombres de variables modifiées, nombre de contraintes, etc) et négligent le point de vue et l'intérêt des acteurs du problème.

Au travers de l'exemple des tournées de véhicules avec fenêtres de temps on présente de nouvelles mesures et l'intérêt qu'elles représentent pour évaluer la proximité de deux solutions. Le problème des tournées de véhicules avec fenêtres de temps fait intervenir trois acteurs : les clients, les livreurs et l'entreprise. Il s'agit pour l'entreprise, de fixer une date de livraison à laquelle devra passer le livreur chez le client. Cette date de livraison doit être choisie dans la fenêtre de temps proposée par le client (heures de présences, etc). Il peut être intéressant pour une entreprise d'être très réactive, c'est-à-dire de traiter les demandes des clients au fur et a mesure qu'elles arrivent et de vouloir les intégrer à la tournée en cours. Selon que l'on se base sur le point de vue de l'entreprise, du client ou du livreur, deux solutions, apriori équivalentes, peuvent être plus ou moins intéressantes. En mêlant dynamicité et différents intervenants, le problème des tournées de véhicules avec fenêtres de temps nous fournit une bonne illustration des problèmes auxquels sont confrontés les utilisateurs.

3.2 Mesures existantes stabilité et dynamisme

Le fait de se baser sur une solution qui est déjà partiellement réalisée implique de vouloir maintenir une certaine continuité dans les nouvelles solutions proposées. On parle alors de stabilité des solutions. Pour que le problème puisse évoluer dynamiquement la modélisation doit permettre la modification des variables et des contraintes. On appelle DCSP une séquence de CSP dans laquelle chaque CSP diffère du précédent par l'ajout ou le retrait d'une contrainte ou d'une variable. On mesure généralement le degré de dynamisme (dod) par :

$$dod = \frac{\text{nombre de variables dynamiques}}{\text{nombre total de variables}}$$

Le degré de dynamisme varie entre 0, lorsque toutes les variables sont statiques (aucune variable n'est modifiée par les évolutions du CSP), et 1, lorsque

toutes les variables sont dynamiques (toutes les variables sont modifiées lors de l'évolution du CSP). La limite de cette mesure est qu'elle ne prend pas en compte l'aspect dynamique de l'arrivée des informations. Or plus l'information arrive tardivement, plus il est difficile de s'adapter efficacement. Le degré de dynamisme effectif est utilisée pour refléter ce côté dynamique . Cette mesure prend en compte : T l'horizon de planification, n_{dyn} le nombre de demandes dynamiques, n_{tot} le nombre de demandes total et tr_i l'instant d'arrivée de la requête i.

$$edod = \frac{\sum_{i=1}^{n_{dyn}} (\frac{tr_i}{T})}{n_{tot}}$$

Ainsi, plus il y aura de variables dynamiques apparaissant tardivement, plus le degré de dynamisme effectif sera proche de 1. Et inversement, lorsque il y a peu de variables dynamiques et que que leurs informations sont connues tôt, le *edod* se rapproche de 0. Cependant cette mesure ne prend pas en compte les fenêtres de temps (intervalle de temps dans lequel une réponse à la requête peut être apportée). Or plus une demande apparaît tardivement (c'est à dire proche de sa date de livraison) dans le problème plus il sera dur de la traiter efficacement (dans le sens où les emplacements possibles pour l'insérer seront limités). Pour prendre en compte cet aspect des choses, on peut proposer une nouvelle mesure $edod'$ où d_i correspond à la date d'exécution, au plus tard, de la requête i et tr_i l'instant d'arrivée de la requête i. Ainsi plus cette mesure est proche de 0, moins il y a de liberté pour placer les sites ajoutés dynamiquement.

$$edod' = \frac{\sum_{i=1}^{n_{dyn}} (\frac{|tr_i - d_i|}{T})}{n_{dyn}}$$

Cette mesure aura tendance à se rapprocher de 0 lorsque les informations seront connues tardivement et à se rapprocher de 1 dans le cas contraire. Puisque une solution a déjà été transmise aux différents acteurs, nous avons intérêt à remettre le moins de choses en cause possible.

3.3 Mesures relatives à l'utilisateur

La stabilité est une propriété qui lie les solutions successives produites par un système dynamique. Une solution est stable, par rapport à la précédente, si le passage d'une solution à l'autre se fait par de petits changements. Une fois ces variations mesurées il faut pouvoir les hiérarchiser. La nuisance provoquée par un changement n'est pas la même, en fonction du point de vue qu'on adopte. Si on se place dans le cadre des tournées de véhicules, un client appréciera que l'heure de passage fixée ne soit pas modifiée alors que pour un chauffeur l'important sera que ses trajets prévus ne varient pas trop. Pour l'entreprise la solution ne soit que légèrement remise en cause afin

que le désagrément créé pour ses clients et employés soit limité. On cherche donc des indicateurs qui permettent d'évaluer l'impact des modifications pour passer d'une solution à l'autre, dans le cadre de problème de tournées de véhicule. Nous considérons des problèmes dynamiques dans lesquels certains sites peuvent apparaître/disparaître d'une solution à l'autre en fonction de l'arrivée de nouvelles commandes ou de l'annulation d'anciennes commandes.

3.3.1 Mesure relative en fonction du point de vue : le cas des tournées de véhicules

Considèrons à titre d'exemple un problème de voyageur de commerce avec fenêtre de temps. On dispose de 2 véhicules. Résoudre ce problème signifie trouver un chemin pour chaque véhicule de telle sorte que l'ensemble des sites soient visités et que la distance totale parcourue par les deux véhicules soit minimale. Dans cet exemple on cherche à résoudre le problème pour 7 sites (le $Site-0$ est le dépôt). Le temps nécessaire pour aller d'un site à un autre est égal à la distance les séparant. Une fois arrivé à un site il faut 5 unité de temps pour réaliser ce qu'il a à faire. Toutes les fenêtres de temps sont fixées à $[0-300]$.

Une solution est représentée par la figure 3.3.1. Le chemin en pointillé correspond au trajet du véhicule 1 et celui en trait plein au véhicule 2. Les valeurs $[t_a, t_d]$ correspondent, respectivement, au temps d'arrivée sur le site et au temps de départ du site par le chauffeur. Ainsi dans l'exemple la solution initiale pour le véhicule-1 est ($Site-0$, $Site-5$, $Site-6$) et pour le véhicule-2 est ($Site-0$, $Site-4$, $Site-3$, $Site-2$, $Site-1$).

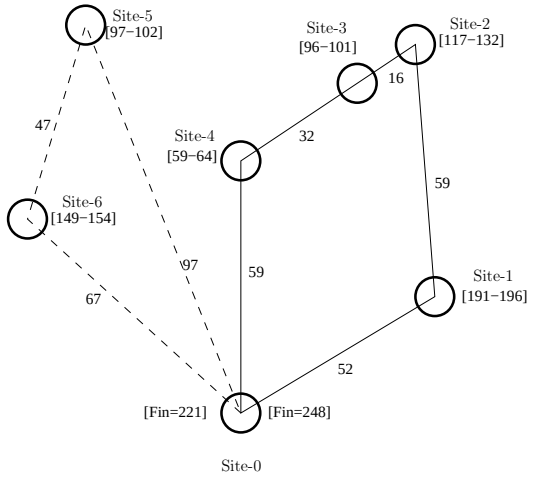

FIGURE 3.1: Solution S_p d'un problème de TSP à 2 véhicules

Peu avant le départ des chauffeurs du dépôt une nouvelle demande est enregistrée (représentée par le $Site-7$). Une nouvelle solution est calculée en

prenant en compte cet ajout (figure 3.3.1). La solution devient alors ($Site-0$, $Site-4$, $Site-7$, $Site-5$, $Site-6$) et ($Site-0$, $Site-3$, $Site-2$, $Site-1$).

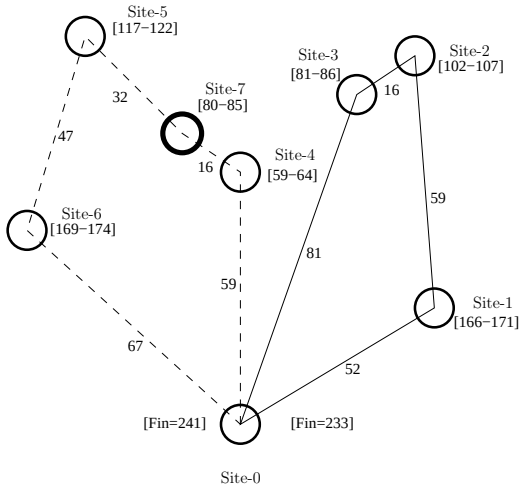

FIGURE 3.2: Solution S_q d'un problème de TSP à 2 véhicules après ajout d'un site

3.3.2 De nouvelles mesures : Point de vue client, point de vue chauffeur.

3.3.2.1 Mesures

Une solution d'un problème de voyageur de commerce est un chemin, de longueur minimale, passant par tous les sites. Le seul acteur sur lequel les variations ont un effet est le chauffeur. La comparaison de deux solutions passe par la comparaison de la position absolue et relative des sites dans le chemin solution. Soient S_p et S_q deux solutions et S_p^i le i^{eme} site visité par la solution p.

- $APosi(S_p, S_q)$ représente le nombre de sites qui ont changé de position dans le chemin solution. On définit α_i par $\alpha_i(S_p, S_q) = 1$ si $S_p^i \neq S_q^i$ et 0 sinon.

$$APosi(S_p, S_q) = \sum_{i \in A}^{n} \alpha_i(S_p, S_q)$$

- $RPosi(S_p, S_q)$ correspond au nombre de sites qui ont changé de position relative. Un site A relativement à site B peut prendre trois positions distinctes : incomparable (si les deux sites ne sont pas desservis par le même véhicule), avant (les deux sites sont desservis par le même véhicule et le véhicule visite A avant B) et après (A et B sont visités par le même véhicule mais A est visité après B). On définit γ_j^i avec $\gamma_j^i(S_p, S_q) = 1$ si

la position relative entre i et j a changé et 0 sinon.

$$RPosi(S_p, S_q) = \sum_{(i,j)\in A\times A, i<j}^{n} \gamma_j^i(S_p, S_q)$$

Les positions relatives n'ont d'importance que pour le chauffeur du véhicule. Or pour le chauffeur ce qui est important c'est le trajet qu'il doit emprunter. Donc, a priori, seule la position relative entre deux sites voisins de la solution est importante. Autrement dit, on considère la route que doit prendre le véhicule pour se rendre au site suivant. On introduit la fonction $Succ(Site - A)$ qui indique à quel site doit aller le chauffeur lorsqu'il est au site $Site - A$.

– $SPosi(S_p, S_q)$ est le nombre de sites dont le successeur a changé. On définit $\omega_i(S_p, S_q)$ tel que $\omega_i(S_p, S_q) = 1$ si $(Succ(S_p[i]) \neq Succ(S_q[i]))$ et 0 sinon.

$$SPosi(S_p, S_q) = \sum_{i\in A}^{n} \omega_i(S_p, S_q)$$

3.3.2.2 Exemple

Ainsi dans le cas de l'exemple précédent $APosi(S_p, S_q) = 6$, $RPosi(S_p, S_q) = 9$ et $SPosi(S_p, S_q) = 3$. Ces mesures traduisent le fait que les premiers sites visités par les chemins ont été modifié ($APosi$ élevé) et que les modifications ont été importantes ($RPosi$ élevé). Cependant la plupart des trajets que devaient effectuer chaque conducteur ont été conservés ($SPosi$ faible).

3.3.3 Stabilité et TSPTW

3.3.3.1 Mesures

Avec l'ajout de la notion de temps et d'heure de passage, de nouvelles comparaisons sont nécessaires pour mesurer l'impact des modifications sur les clients. Soit t_i^k date de début de la tâche i dans la solution k.

– $Diff(S_p, S_q)$ est le nombre de sites qui ne sont plus visités à la même date. On définit δ_i tel que $\delta_i(S_p, S_q) = 1$ si $(t_i^p \neq t_i^q)$ et $\delta_i(S_p, S_q) = 0$ sinon. Alors :

$$Diff(S_p, S_q) = \sum_{i\in A}^{n} \delta_i(S_p, S_q)$$

– $Sdif(S_p, S_q)$ est la somme des perturbations :

$$Sdif(S_p, S_q) = \sum_{i\in A}^{n} |t_i^p - t_i^q|$$

– $Mdif(S_p, S_q)$ est la perturbation maximale :

$$Mdif(S_p, S_q) = max_{i \in A}^{n}\{|t_i^p - t_i^q|\}$$

Les deux dernières mesurent l'importance des changements mais elles ne permettent pas d'évaluer l'impact direct sur les acteurs intervenant dans le problème.

3.3.3.2 Exemple

Si on reprend l'exemple initial : $Diff(S_p, S_q) = 6$, $Sdiff(S_p, S_q) = 85$ et $Mdif(S_p, S_q) = 20$. Donc beaucoup de clients vont être dérangés par les modifications ($Diff$ élevée) mais le dérangement va être faible (valeur de $Mdif$ et $Sdiff$).

3.3.4 Stabilité et TSP multi-véhicule

3.3.4.1 Mesures

Lorsqu'on considère plus d'un véhicule se pose aussi la question de savoir si la nouvelle solution nécessite qu'un chauffeur se charge d'un client qui ne lui était initialement pas affecté. Et en cas de changement de combien la durée de sa tournée va être modifiée.

– $Vehi(S_p, S_q)$ correspond au nombre de sites qui changent de véhicule d'une solution à l'autre. On définit ν_i tel que $\nu_i(S_p, S_q) = 1$ si t_i^p et t_i^q sont dans le même véhicule et $\nu_i(S_p, S_q) = 0$ sinon.

$$Vehi(S_p, S_q) = \sum_{i \in A, i}^{n} \nu_i(S_p, S_q)$$

– $dVehi(S_p, S_q)$ représente le temps de dérangement total.

$$dVehi(S_p, S_q) = \sum_{j \in V}^{m} |t_{V_j^p} - t_{V_j^q}|$$

3.3.4.2 Exemple

$Vehi(S_p, S_q) = 1$ et $dVehi(S_p, S_q) = 15$. Ces mesures permettent de préciser les mesures précédentes.

3.3.5 Utilisation des mesures

Au travers des différentes mesures introduites on voit apparaitre la notion de mesure contextuelle. C'est à dire de mesures qui prennent en comptes la façon dont sont "vécues" les perturbations par les différents acteurs du problème. Elles peuvent prendre en compte le changement de valeurs des

différentes variables (écart avec les valeurs prévues) ou les changements dans la topologie des solutions (forme de la solutions, etc). Ces mesures ont avant tout pour but de réintroduire de la sémantique dans la solution afin d'interpréter les variations dans leurs contextes en fonction de ce que représentent les différentes variables et valeurs.

Ces mesures peuvent être utilisées de deux manières. Elles peuvent servir pour orienter la prise de décision du décideur. Elles peuvent être utilisées pour guider la recherche et favoriser les solutions proches de la solution initiale.

3.3.5.1 Utilisation par le décideur

Si on reprend l'exemple des tournées de véhicules, le but de l'entreprise est de faire des bénéfices tout en satisfaisant au maximum ses clients et son personnel. Ainsi pour l'entreprise le choix va dépendre de l'aspect qu'elle veut privilégier : les gains, la satisfaction des clients, le confort des livreurs. La question de la stabilité intervient donc de manière indirecte. Si on ajoute au problème la gestion des stocks, la stabilité peut survenir au travers de l'impact des perturbations sur l'état des stocks (surplus, pénurie). Les mesures en fonction des différents point de vue permettent donc au décideur d'orienter son choix.

3.3.5.2 Utilisation durant la recherche d'une solution

Comme indiqué dans l'introduction, une solution est définie comme stable si le passage d'une solution à une autre est possible en effectuant peu de changements. Donc la propriété de stabilité est fortement liée à la topologie de l'espace de recherche : plus les bassins d'attractions sont larges, plus il est facile de trouver une solution voisine (faisable et proche de l'optimum). A priori, dans ce cas une recherche locale "classique" a de fortes chances de trouver rapidement une autre solution de bonne qualité.

Shaw a montré que la PPC (utilisée lors d'une recherche locale) devient intéressante dès que l'on considère des problèmes possédant de nombreuses contraintes additionnelles. Le problème, a priori, c'est que dans un problème fortement contraint l'espace de recherche a peu de chance de posséder une topologie propice à la stabilité et une recherche locale explorant de manière aveugle (guidée uniquement par la faisabilité et la qualité des solutions, la stabilité étant un effet de bord) risque de très vite s'éloigner de la solution initiale. Il est donc intéressant d'utiliser ces mesures comme heuristiques de choix de variables/valeurs.

La stratégie d'exploration de l'espace de recherche basée sur *LDS* [HG95] permet d'explorer les affectations en fonction de leur distance par rapport à la solution de référence. En effet, *LDS* consiste à parcourir l'espace de recherche en s'éloignant progressivement d'une solution idéale. L'algorithme *LDS* est donné dans les figures 2 et 1. *Successors* est une fonction qui re-

tourne une liste de 0 ou 2 variables en fonction des préférences de l'heuristique. x représente la distance à laquelle la recherche peut s'éloigner d'une solution "idéale" selon l'heuristique. On appelle *LDS-Probe* itérativement, en incrémentant x à chaque fois. *LDS-Probe* réalise une recherche en profondeur et limitant le nombre de divergences à x. Lorsque x atteint d, la profondeur maximale de l'arbre, *LDS-Probe* parcours l'intégralité de l'arbre. Ainsi la recherche garantit de trouver une solution si il en existe une et de terminer si il n'en existe pas. Cependant, puisque *LDS-Probe* limite le nombre de divergences à x (et non pas à exactement x divergences), l'itération n réexamine les nœuds considérés durant les itérations précédentes. Les mesures de distance présentées plus haut peuvent être utilisées dans la fonction Successors afin de sélectionner les nœuds "proches" de la solution courante.

Algorithm 1 LDS-Probe(*node*, k)

```
1: if Goal-P(node) then
2:    return node;
3: end if
4: s := Successors(node);
5: if Null − P(node) then
6:    return Nil;
7: end if
8: if k = 0 then
9:    return LDS-Probe(First(s), 0);
10: else
11:    result := LDS-Probe(Second(k), k − 1);
12:    if result ≠ Nil then
13:       return result;
14:    end if
15:    return LDS-Probe(First(s), k);
16: end if
```

Algorithm 2 *LDS*(*node*)

```
1: for x = 0 to maximum length do
2:    result := LDS-Probe(node, x);
3:    if result ≠ Nil then
4:       return result;
5:    end if
6: end for
7: return Nil;
```

3.4 Conclusion

Il faut replacer les problèmes dynamiques dans leur contexte. En effet les solutions successives peuvent avoir déjà été mises partiellement en application lorsqu'un changement survient. Il faut alors être capable de prendre en compte l'impact des changements sur les différents intervenants, c'est-à-dire introduire du sens dans la mesure de stabilité.

Au travers de l'exemple des tournées de véhicules, on introduit différentes mesures possibles pour évaluer la stabilité d'une solution en fonction du point de vue. Les mesures peuvent dépendre : soit de la variation absolue des éléments dans la solution, soit de la variation relative des éléments les uns par rapport aux autres. On parle alors de mesures contextuelles. Il s'agit de quantifier la nuisance subit par les différents acteurs pour permettre une prise de décision prenant en compte les intérêts de tous les acteurs.

Une seconde utilisation est l'intégration des mesures comme heuristiques afin d'orienter la recherche et de découvrir des solutions potentiellement stables. Cependant l'utilisation d'heuristiques n'empêche pas le travail redondant. Nous allons nous concentrer, dans la deuxième partie, sur différentes techniques qui permettent de limiter l'exploration de zones de l'espace de recherche déjà visitées.

Deuxième partie

Les CSP dynamiques

Chapitre 4

État de l'art - Des CSP aux DCSP

Les problèmes que l'on rencontre dans la vie courante peuvent généralement se modéliser sous la forme d'un ensemble de variables (représentant les caractéristiques du problème) devant vérifier une ou plusieurs contraintes. Une contrainte correspond aux propriétés que l'objet doit vérifier, que ce soit des propriétés inhérentes à la nature des variables ou des propriétés voulues par l'utilisateur. Ainsi un grand nombre de problèmes peut être représenté grâce aux réseaux de contraintes, c'est-à-dire par un ensemble de contraintes portant sur des variables.

Pour être au plus proches des problèmes réels les réseaux de contraintes doivent aussi prendre en compte l'aspect dynamique. Ainsi après avoir introduit la programmation par contraintes, nous présenterons différentes approches proposées pour intégrer la dimension dynamique aux réseaux de contraintes et différentes manière de les traiter.

4.1 La programmation par contraintes

Les contraintes apparaissent dans la plupart des activités humaines. Elles formalisent les dépendances présentes dans le monde physique de manière naturelle et transparente. Une contrainte est une relation logique entre plusieurs variables, chacune prenant une valeur dans un domaine donné. Ainsi les contraintes restreignent les valeurs que peuvent prendre les variables. Elles peuvent aussi lier des variables hétérogènes et lier des variables de domaines différents (entier, bit, ...). Pour l'utilisateur un des avantages des contraintes c'est le fait qu'elles soient déclaratives, c'est-à-dire qu'elles spécifient la relation liant différentes variables sans spécifier l'algorithme à utiliser pour assurer le maintien de cette propriété. En pratique, un utilisateur n'a donc à se soucier que de la manière dont il veut modéliser son problème sans se poser la question de la résolution.

Dans la vie de tous les jours, nous utilisons naturellement les contraintes pour guider le raisonnement. Mais la difficulté des problèmes depend du nombre de contraintes que nous avons à considérer et l'interdépendance existant entre les différentes contraintes. Le but de la programmation par contraintes est l'étude des systèmes basé sur les contraintes. L'idée est de résoudre des problèmes en posant des contraintes et en cherchant une solution satisfiant l'ensemble des contraintes.

Les origines de la programmation par contraintes se retrouvent dans l'Intelligence Artificielle qui exploite un ensemble de contraintes et la Programmation Logique qui repose sur un paradigme déclaratif. De nos jours les applications réelles de la programmation par contraintes sont dans le domaine de la planification, de l'ordonnancement et l'optimisation auquels la Recherche Opérationnelle propose aussi des solutions. La Recherche Opérationnelle a été l'objet d'études depuis des années et a apporté des réponses très efficaces à certains problèmes. À la différence de la programmation linéaire, la program-

mation par contraintes permet de proposer des modèles de haut niveau et de trouver des solutions plus facilement compréhensible par l'utilisateur. De plus la programmation par contraintes permet de faire coopérer des approches de domaines différents.

Les problèmes auxquels s'intéressent la programmation par contraintes sont modélisés sous la forme de problème de satisfaction de contrainte (Constraint satisfaction problems). Les CSP ont été étudiés durant de nombreuses années en Intelligence Artificielle. Un CSP est définit par :

- un ensemble de variables
- pour chaque variable, un domaine fini de valeurs possible pour la variable
- un ensemble de contrainte restreignant les valeurs que les variables peuvent prendre simultanément

Le but est de trouver une solution c'est à dire : une affectation de valeur pour chaque variable, de telle sorte que toutes les contraintes soient satisfaites. En fonction des besoins de l'utilisateur on peut rechercher :

- une solution (sans préférences sur la forme de la solution à trouver)
- toutes les solutions
- une solution optimale ou au moins une bonne solution, en donnant une fonction objective portant sur une partie ou la totalité des variables

Les solutions d'un CSP peuvent être trouvées en cherchant (systématiquement) parmi les affectations possibles des valeurs. D'un point de vue théorique, résoudre un CSP en utilisant l'exploration systématique est assez simple. Mais d'un point de vue pratique, la difficulté consiste à trouver des méthodes d'exploration efficaces.

Même si les méthodes de recherche systématique peuvent paraître simples et peu efficaces, elles servent de base à l'élaboration de méthodes plus évoluées. La méthode d'exploration la plus basique, qui explore l'espace de recherche, est *generate-and-test*. L'idée de *generate-and-test* est de générer, dans un ordre aléatoire, la totalité des affectations possibles. Si un des affectations satisfait l'ensemble des contraintes alors une solution est trouvée. L'efficacité de *generate-and-test* est très mauvaise puisque la génération est aveugle et la détection des inconsistances ne se fait qu'au dernier moment. Les deux pistes pour améliorer les deux principaux défauts de la méthode *generate-and-test* sont donc :

- utiliser un générateur d'affectations qui prend en compte les conflits pour converger plus rapidement vers une solution
- mélanger le générateur et le testeur afin que la validité des contraintes soit testée dès l'instanciation des variables. Le but est de pouvoir revenir, au plus tôt, sur les affectations inconsistantes : (*backtracking*).

Le *backtracking* est utilisée dans la résolution des CSP afin de remettre en cause les décisions prises. On parle de *backtracking* chronologique car on revient en arrière sur les dernières décisions prises. Il repose sur l'extension, de manière incrémentale, de la solution partielle courante en lui ajoutant une valeur consistante (n'entrainant pas de conflit avec une contrainte). Ainsi les variables sont fixées de manière séquentielle et lorsque toutes les variables participant à une contrainte sont fixées, la validité de la contrainte est vérifiée. Si une solution partielle viole une contrainte, un retour en arrière sur le dernier choix réalisé est effectué. Il y a trois inconvénients majeurs au *backtracking* chronologique :

- *thrashing* : un échec répété du à la même cause
- le travail redondant : les valeurs conflictuelles des variables ne sont pas enregistres
- la détection tardive des conflits : les conflits ne sont pas détectés avant de survenir

Des solutions ont été proposée pour répondre aux deux premiers inconvénients au travers du *backjumping* [Pro93] et du backmarking mais les efforts ont surtout été concentrés sur la détection des inconsistances des solutions partielles.

Ainsi, la méthode de base d'exploration de l'arbre de recherche (Backtrack chronologique) peut être améliorée de deux façons. La première manière consiste à supprimer le plus tôt possible les valeurs inconsistantes afin de réduire les domaines des variables. On utilise pour cela des algorithmes de filtrages (FC, MAC...) qui utilisent les propriétés du CSP pour faire le maximum possible de déductions sur la consistance. Cela permet, a priori, de découvrir rapidement les branches qui ne contiennent pas de solutions et évite donc de passer trop de temps à explorer des branches sans intérêt. La deuxième façon est d'utiliser les informations tirées des branches déjà explorées afin de pouvoir détecter les affectations pertinentes et diriger au mieux l'exploration des autres branches (CBJ, DBT...). On parle alors de méthodes rétrospectives. Les méthodes rétro-prospectives (FC-CBJ, MAC-CBJ, MAC-DBT [JB97],...) correspondent à l'utilisation conjointe de ces deux améliorations afin d'obtenir des algorithmes aussi efficaces que possible.

Les méthodes prospectives reposent sur la considération suivante : une valeur qui est localement inconsistante ne peut pas être complétée en une affectation globale consistante. On peut donc retirer du domaine de définition des différentes variables toutes les valeurs localement inconsistante sans perdre de solutions. L'objectif est donc de détecter au plus tôt les affectations partielles qui ne pourront pas être prolongées. Dans ce but les méthodes prospectives propagent les modifications faites à une variable (affectation, suppression d'une valeur...) afin de filtrer les valeurs des domaines. Cela permet une détection précoce des branches inintéressantes sans modifier les sens de par-

cours de l'arbre de recherche qui reste une exploration en profondeur d'abord.

La propagation peut se faire par l'intermédiaire de différentes propriétés : consistance de nœud, consistance d'arc, consistance de chemin...

Ainsi les méthodes de résolution sont conçues pour prendre en compte des modèles statiques. Plusieurs solutions ont été proposé pour prendre en compte des problèmes dynamiques à l'aide de modèles statiques. Nous allons en présenter quelques uns en reprenant en partie l'état de l'art présent dans [VJ05].

4.2 Présentation des DCSP

L'intérêt des réseaux de contraintes pour structurer la connaissance et modéliser des environnements dynamiques a été mis pour la première fois en évidence dans [DD88]. Les auteurs définissent un réseau dynamique de contraintes comme une séquence de réseaux statiques de contraintes dans laquelle chaque réseau de contrainte est produit en modifiant le réseau précédent. L'objectif est de pouvoir prendre en compte les perturbations subies par l'environnement en modifiant dynamiquement le modèle utilisé (par l'ajout/-suppression de contraintes, variables et valeurs). Comme conséquence de ces changements incrémentaux, l'ensemble des solutions du CSP (définition 1 et définition 2) considéré peut diminuer (restriction définition 3) ou augmenter (relaxation définition 4). Les auteurs se limitent à des CSP binaires et proposent une solution dans le cas où le réseau de contraintes est acyclique.

Définition 1. Un CSP (Constraint Satisfaction Problem) est un tuple $\langle \mathcal{C}, \mathcal{V}, \mathcal{D} \rangle$ tel que $\mathcal{V} = \langle v_1, \ldots, v_n \rangle$ est un ensemble de variables, $\mathcal{D} = \langle D_1, \ldots, D_n \rangle$ est l'ensemble des domaines des variables avec D_i le domaine de la variable v_i. On note $dom(v_i) = D_i$ et $var(D_i) = v_i$. $\mathcal{C} = \langle c_1, \ldots, c_m \rangle$ est l'ensemble des contraintes. On note $constraint(v_i)$ l'ensemble de contraintes portant sur la variable v_i.

Définition 2. On dit que $\mathcal{S} = \langle x_1, x_2, \ldots, x_n \rangle$ est une solution du CSP $= \langle \mathcal{C}, \mathcal{V}, \mathcal{D} \rangle$ si et seulment si $\forall i \in [1, |\mathcal{V}|], x_i \in dom(v_i)$ et $\mathcal{S}$ consistante avec l'ensemble des contraintes $\mathcal{C}$.

Définition 3. Une restriction du problème $\mathcal{P}$ est un problème $\mathcal{P}'$ tel que $\mathcal{S}' \subseteq \mathcal{S}$ où $\mathcal{S}$ est l'ensemble des solutions du problème $\mathcal{P}$ et $\mathcal{S}'$ l'ensemble des solutions du problème $\mathcal{P}'$.

Définition 4. Une relaxation du problème $\mathcal{P}$ est un problème $\mathcal{P}'$ tel que $\mathcal{S}' \supseteq \mathcal{S}$ où $\mathcal{S}$ est l'ensemble des solutions du problème $\mathcal{P}$ et $\mathcal{S}'$ l'ensemble des solutions du problème $\mathcal{P}'$.

Dans le contexte binaire (les contraintes portent sur, au plus, deux variables), une restriction survient lorsqu'une nouvelle contrainte est posée sur

un sous-ensemble de variables existant, ou lorsqu'une nouvelle variable est ajoutée au problème par l'intermédiaire de nouvelles contraintes. Les restrictions étendent toujours le modèle que ce soit par l'ajout de variables ou par l'ajout de contraintes. Les relaxations surviennent lorsque des contraintes qui s'appliquaient jusqu'à présent deviennent invalides et doivent être retirées du réseau.

La définition d'un réseau dynamique de contraintes proposée par Dechter dans le cadre binaire peut être étendue aux contraintes n-aires.

Définition 5. Un DCSP $\mathcal{P}$ est une séquence $\mathcal{P}_0, \ldots, \mathcal{P}_n$ de CSP statiques telle que $\mathcal{P}_{i+1} = \mu(\mathcal{P}_i)$ où $\mu : CSP \longrightarrow CSP$.

Dans ce contexte on peut se demander ce que signifie résoudre un DCSP. Est ce qu'il s'agit de trouver une séquence de solutions ? Une solution compatible avec au moins un des CSP de la séquence ? Avec l'ensemble des CSP ? La subtilité des DCSP est que la dimension dynamique du problème n'est pas modélisée dans le CSP mais est intégrée à la méthode de résolution. Par exemple, plaçons nous dans le contexte d'un problème de tournées de véhicules dans lequel de nouvelles demandes apparaitraient.

Initialement, la personne chargée de la planification a connaissance d'un ensemble de commande à satisfaire. Il modélise alors le problème en prenant en compte les informations disponibles et cherche une solution. Une fois qu'une solution répondant à ses critères a été trouvée, il la transmet aux chauffeurs qui la mette en application. En fonction de la réactivité souhaitée, différents cas de figures peuvent survenir lorsqu'une nouvelle commande est reçue. Dans un système classique non-réactif, toute solution qui a commencé a être mise en application est réalisée jusqu'à son terme. Les nouvelles commandes reçues entre temps seront alors traitées lors de la planification de la période suivante. Dans un système nécessitant une plus grande réactivité, il peut être intéressant de prendre en compte le plus tôt possible, les nouvelles commandent qui arrivent, ce qui implique de pouvoir remettre en cause la solution en cours. Dans un système nécessitant une très grande réactivité, même l'étape de recherche de solution ne peut plus être considéré comme un élément ponctuel. Ainsi une nouvelle commande peut interrompre la résolution.

Une fois le contexte replacé, la réponse à la question : que signifie résoudre un DCSP est plus évidente. Puisque les différents CSP de la séquence correspondent à la connaissance disponible à une date t, l'objectif est de trouver une solution au CSP le plus récent (le dernier de la séquence). Du fait du caractère incremental de la séquence, les solutions des autres CSP de la séquence peuvent être des solutions intermédiaires intéressantes. Ainsi l'efficacité des DCSP reposent sur l'utilisation de méthodes permettant d'exploiter efficacement le travail effectué aux étapes antérieures.

Pour savoir comment réutiliser le travail déjà fait nous allons nous intéresser aux transformations possibles pour passer d'un CSP_i à un $CSP_{i+1} = \mu(CSP_i)$ et aux conséquences de ces transformations sur l'ensemble

des solutions. On indentifie 6 transformations élémentaires (Figure 4.5) telles que toute transformation μ peut s'exprimer comme une composition de ces transformations.

Démonstration. Soient $\mathcal{P}$ et $\mathcal{P}'$ tel que $\mathcal{P}' = \mu(\mathcal{P})$ avec $\mathcal{P} = \langle \mathcal{C}, \mathcal{V}, \mathcal{D} \rangle$ et $\mathcal{P}' = \langle \mathcal{C}', \mathcal{V}', \mathcal{D}' \rangle$. On note $\mathcal{D}_a$ l'ensemble des tuples (v, val) ajoutés à $\mathcal{D}$ tel que $\mathcal{D} \cap \mathcal{D}_a = \emptyset$ et $\mathcal{D} \cup \mathcal{D}_a = \mathcal{D}'$. On note $\mathcal{D}_r$ l'ensemble des tuples (v, val) retirés de $\mathcal{D}$ tel que $\mathcal{D} \cup \mathcal{D}_r = \mathcal{D}_a$ et $\mathcal{D} \subset \mathcal{D}_r = \mathcal{D}'$. On définit de la même maniere, l'ensemble des contraintes ajoutées $\mathcal{C}_a$, l'ensemble des contraintes retirées $\mathcal{C}_r$, l'ensemble des variables ajoutées $\mathcal{V}_a$ et l'ensembles des variables retirées $\mathcal{V}_r$.

Soit

$$\begin{aligned} \mu' \quad = \quad & (\circ_{v \in \mathcal{V}_a} \mu_v) \circ (\circ_{D \in \mathcal{D}_r} \mu_{\neg D}) \circ \\ & (\circ_{D \in \mathcal{D}_a} \mu_D) \circ (\circ_{c \in \mathcal{C}_r} \mu_{\neg c}) \circ \\ & (\circ_{c \in \mathcal{C}_a} \mu_c) \circ (\circ_{v \in \mathcal{V}_r} \mu_{\neg v}) \end{aligned}$$

Ainsi

$$\begin{aligned} \mu'(\mathcal{P}) \quad = \quad & \langle (\mathcal{C} \cup \mathcal{C}_a) \setminus (\mathcal{C}_r \cup_{var \in \mathcal{V}_r} constraint(var)), \\ & (\mathcal{V} \cup \mathcal{V}_a) \setminus \mathcal{V}_r, \\ & (\mathcal{D} \cup \mathcal{D}_a) \setminus (\mathcal{D}_r \cup_{var \in \mathcal{V}_r} dom(var)) \rangle \\ = \quad & \langle (\mathcal{C} \cup \mathcal{C}_a) \setminus \mathcal{C}_r, \\ & (\mathcal{V} \cup \mathcal{V}_a) \setminus \mathcal{V}_r, \\ & (\mathcal{D} \cup \mathcal{D}_a) \setminus \mathcal{D}_r \rangle \\ = \quad & \langle \mathcal{C}', \mathcal{V}', \mathcal{D}' \rangle \\ = \quad & \mathcal{P}' \end{aligned}$$

car $\mathcal{D}_r \supseteq \cup_{var \in \mathcal{V}_r} D_{var}$ et $\mathcal{C}_r \supseteq \cup_{var \in \mathcal{V}_r} constraint(var)$,d'où $\mu = \mu'$. □

Transformation de $\mathcal{P}$ à $\mathcal{P}'$	μ	$\mathcal{C}'$	$\mathcal{D}'$	$\mathcal{V}'$
Ajout d'une contrainte c	μ_c	$\mathcal{C} \cup c$	$\mathcal{D}$	$\mathcal{V}$
Retrait d'une contrainte c	$\mu_{\neg c}$	$\mathcal{C} \setminus c$	$\mathcal{D}$	$\mathcal{V}$
Ajout d'une valeur val	$\mu_{(v_j,val)}$	$\mathcal{C}$	$\mathcal{D} \cup (v_j, val)$	$\mathcal{V}$
Retrait d'une valeur val	$\mu_{\neg(v_j,val)}$	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{V}$
Ajout d'une variable var	μ_{var}	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{V} \cup var$
Retrait d'une variable var	$\mu_{\neg var}$	$\mathcal{C} \setminus (\cup_{c \in c(var)} c)$	$\mathcal{D} \setminus D_{var}$	$\mathcal{V} \setminus var$

FIGURE 4.1: Transformations μ élementaires, tel que $\mathcal{P}' = \mu(\mathcal{P})$ avec $\mathcal{P} = \langle \mathcal{D}, \mathcal{V}, \mathcal{C} \rangle$ et $\mathcal{P}' = \langle \mathcal{D}', \mathcal{V}', \mathcal{C}' \rangle$.

Il faut différencier les transformations élémentaires qui correspondent aux transformations réalisables dans un CSP des transformations usuelles, c'est-à-dire des transformations qui ont un sens dans le cadre de l'évolution du modèle. En effet du point de vue sémantique l'ajout d'une variable n'a de sens que s'il est associé à l'ajout de contraintes portant sur cette variable.

Le but d'un DCSP est de permettre la prise en compte des perturbations qui surviennent en modifiant le modèle. Ainsi en pratique, seuls quelques types de transformations sont effectivement utilisés pour passer d'un problème $\mathcal{P}_i$ à un problème $\mathcal{P}_{i+1}$: ajouter une contrainte (μ_c), retirer une contrainte ($\mu_{\neg c}$), ajouter une valeur ($\mu_{(v_j,val)}$), retirer une valeur ($\mu_{\neg(v_j,val)}$), ajouter une variable ($(\bigcirc_{c\in\mathcal{C}_{var}}\mu_c)\circ\mu_{var}$) et retirer une variable ($(\bigcirc_{c\in\mathcal{C}_{var}}\mu_c)\circ\mu_{\neg var}$).

La difficulté supplémentaire liée au cas n-aire est que l'ajout d'une variable peut remettre en cause des contraintes préexistantes. Ainsi contrairement au cas binaire ou ajouter une variable correspond à une restriction du problème (puisque l'ajout d'une variable se traduit uniquement par l'ajout de contraintes) l'effet de l'ajout d'une variable sur l'ensemble des solutions est indéterminé. On peut cependant décomposer chaque modification μ en une séquence de modifications correspondant soit à une restriction, soit à une relaxation.

Chaque transformation d'un problème $\mathcal{P}_i$ à un problème $\mathcal{P}_{i+1}$ peut être décomposée en une suite commutative de transformations correspondant à des restrictions ou à des relaxations. Ainsi pour pouvoir exploiter un DCSP il faut être capable de calculer le nouvel ensemble des solutions lors d'une restriction et lors d'une relaxation. En fonctions des méthodes utilisées certains ordres dans la suite de la décomposition peuvent être plus intéressante : regrouper les transformations correspondant à des restrictions, regrouper les transformations portant sur une même contrainte...

L'ensemble des modifications possibles d'un CSP (ajout ou retrait de variables, valeurs, contraintes) peuvent être exprimés comme des ajouts (restrictions) et retraits (relaxations) de contraintes. En effet les domaines peuvent être vus comme des contraintes unaires, retirer ou ajouter une variable revient à modifier toutes les contraintes dans lesquelles la variable participe et modifier une contrainte revient à retirer puis rajouter la contrainte.

Nous venons de présenter la définition de base d'un CSP dynamique et les façons dont il peut évoluer dans le temps. Nous allons maintenant nous attarder sur le lien qu'il existe entre les DCSP et les autres formalismes existants pour gérer l'aspect dynamique d'un problème.

4.3 DCSP et autres formalismes

Il faut différencier le formalisme des DCSP de celui des CSP qui modélisent un problème dynamique. En effet, le but du DCSP est d'adapter un CSP aux

perturbations extérieures venant modifier le problème et non pas de prendre en compte des évolutions déjà intégrées au modèle. On reprend en partie différents modèles, en les précisant, de CSP dédiés aux problèmes dynamiques présenté dans [VJ05].

Transformation de $\mathcal{P}$ à $\mathcal{P}'$	μ	$\mathcal{C}'$
Ajout d'une contrainte c	μ_c	restriction
Retrait d'une contrainte c	$\mu_{\neg c}$	relaxation
Ajout d'une valeur val	$\mu_{(v_j,val)}$	relaxation
Retrait d'une valeur val	$\mu_{\neg(v_j,val)}$	restriction
Ajout d'une variable var	μ_{var}	relaxation
Retrait d'une variable var	$\mu_{\neg var}$	?

FIGURE 4.2: Effets des transformations μ élementaires, tel que $\mathcal{P}' = \mu(\mathcal{P})$ avec $\mathcal{P} = \langle \mathcal{D}, \mathcal{V}, \mathcal{C} \rangle$ et $\mathcal{P}' = \langle \mathcal{D}', \mathcal{V}', \mathcal{C}' \rangle$, sur l'ensemble des solutions.

4.3.1 TCSP

Le TCSP (Temporal CSP) décrit dans [SD91] est un CSP permettant de prendre en compte le temps. Les auteurs s'appuient sur l'exemple suivant : la date d'aujourd'hui est le 1er janvier. La tâche est d'acheminer un cargo de New York à Los Angeles pour le charger soit entre le 8 et le 10 janvier, soit entre le 15 et 16 ou entre le 18 et 19. Le cargo doit voyager entre Chicago et Dallas. En utilisant des transports aériens le cargo peut être acheminé de New York à Chicago en 1 ou 2 jours, en utilisant des transports terrestres cela prend de 10 à 11 jours. Pour aller de Chicago à Los Angeles il faut 3 à 4 jours en utilisant plusieurs transports aériens tandis que par transport terrestre il faut entre 13 et 15 jours [...]. Leur but est de pouvoir répondre à des questions du type : "Quand est-ce que le cargo peut arriver à Dallas ?" ou "Est-ce que le cargo pourra être à Los Angeles le 18 et 19 janvier ?". Les auteurs définissent plusieurs contraintes correspondantes aux différentes relations temporelles qu'il peut exister entre un événement ponctuel et un événement long (avant, commence, pendant, finit, après) et entre deux événements longs (avant, après, à la suite, recouvre, pendant, contient, égaux, distincts, commencent, finissent). Ils proposent ensuite plusieurs algorithmes pour filtrer ce type de CSP.

4.3.2 Le OCSP

Les techniques de satisfaction de contraintes ont été utilisé avec succès pour traiter des problèmes d'allocation de ressources, de planification, d'ordonnancement, ainsi que pour des problèmes de configuration. Traditionnellement, ces problèmes sont traités dans des mondes fermés : toutes les variables, domaines, contraintes et relations sont définis et connus avant la résolution.

Dans [FMG05], les auteurs définissent les OCSP (Open CSP), c'est à dire des CSP dans lesquels les domaines des variables ne sont pas, initialement, totalement connus. Leur objectif est de pouvoir prendre en compte des systèmes distribués dans lesquels connaitre les valeurs des domaines passent par des requêtes coûteuses à des serveurs distants (Figure 4.3).

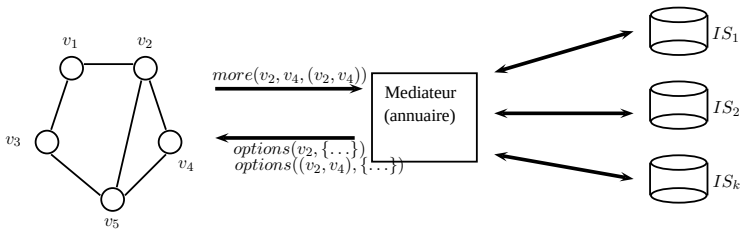

FIGURE 4.3: Open CSP. Le solveur peut accéder aux informations disponibles sur les serveurs grâce au médiateur. En envoyant le message $more(v_i, \dots, (v_i, v_j), \dots)$ le solveur peut demander au médiateur de récupérer plus de valeurs pour ces variables. Le médiateur serait un serveur indexant l'information disponible sur les serveurs d'information. A l'aide des requêtes envoyées le solveur pourrait ainsi découvrir de nouvelles valeurs dans les domaines de ses variables ou de nouvelles relations entre ses variables. En utilisant les messages $options(v_i, \dots)$ et $options((v_i, v_j), \dots)$ le médiateur informe le solveur des valeurs et des tuples interdits découverts sur le réseau. Le médiateur retourne $nomore(v_i, \dots)$ pour informer le solveur que plus aucune nouvelle valeur peut être trouvée pour une variable.

Ce principe remet en cause les techniques habituelles utilisées pour la propagation de contraintes et le filtrage. Dans un tel contexte, rendre une méthode de recherche complète et terminale est beaucoup plus complexe. Le fait que l'exploration des domaines soit coûteuse, fait aussi apparaitre de nouvelles préoccupations, comme la nécessité d'avoir des heuristiques permettant de minimiser le nombre de requêtes à effectuer pour trouver une solution.

Formellement les auteurs définissent l'OCSP comme :

Définition 6. Un OCSP est un ensemble, éventuellement non-borné, partiellement ordonné $\{CSP(0), CSP(1), \dots\}$ de problèmes de satisfaction de contraintes, où $CSP(i)$ est défini par un tuple $< X, C, D(i), R(i) >$ avec :

1. $X = \{x_1, x_2, \dots, x_n\}$ est un ensemble de n variables,
2. $C = \{(x_i, x_j, \dots), (x_k, x_l, \dots), \dots\}$ est un ensemble de m contraintes, données par l'ensemble ordonné des variables qu'elles impliquent,
3. $D(i) = \{d_1(i), d_2(i), \dots, d_n(i)\}$ est l'ensemble des domaines pour le $CSP(i)$ avec $d_k(0) = \emptyset, \forall k$.
4. $R(i) = \{r_1(i), r_2(i), \dots, r_m(i)\}$ est l'ensemble des relations pour le $CSP(i)$, chacun donnant la liste des tuples autorisés pour les variables impliquées dans la contrainte correspondante et vérifiant $r_k(0) = \emptyset, \forall k$.

L'ensemble est ordonné par la relation $\prec$ où $CSP(i) \prec CSP(j)$ si et seulement si $(\forall k \in [1 \dots n])d_k(i) \subseteq d_k(j), (\forall k \in [1 \dots m])r_k(i) \subseteq r_k(j), etsoit(\exists k \in [1 \dots n]d_k(i) \subset d_k(j))ou(\exists k \in [1 \dots m]r_k(i) \subset r_k(j))$.

Une solution d'un OCSP est une combinaison d'affectation de valeurs à toutes les variables telle que pour un i, chaque valeur appartiennent au domaine correspondant et chaque combinaison de valeurs appartienne à relations correspondantes du $CSP(i)$.

Dans [vHR06] les auteurs donnent des exemples concrets de problèmes nécessitant l'utilisation d'open-constraint dans des mondes fermés, c'est à dire de contraintes dont l'ensemble des variables sur lesquelles elles portent n'est pas connu au début de la résolution. Ils se concentrent en particulier sur le cas de la contrainte open-gcc. Considérons un ensemble d'activités et supposons que chaque activité peut être réalisée soit sur la ligne 1 formée par l'ensemble des ressources unaires R_1 ou sur la ligne 2 formé par l'ensemble des ressources unaire R_2 (Figure 4.4). Au début, les ressources qui vont être utilisées par une activité sont inconnues. Aussi, l'ensemble des activités qui vont être traitées par une ressource est inconnu. Cependant il est utile de pouvoir exprimer que les activités qui vont être traitées sur chaque ligne doivent être deux a deux distinct. Ceci peut être assuré en définissant deux contraintes alldifferent impliquant les variables de début de chaque activité et en exprimant que une variable de début doit être impliqué dans exactement une contrainte alldifferent. Initialement, chaque contrainte alldifferent est définie sur un ensemble formé par toutes les variables de départ. L'ensemble est ensuite modifié lorsqu'il peut être prouvé qu'une variable ne peut pas être membre de l'une des contrainte alldifferent (l'activité ne peut pas être traitée par la ligne de production) ou que la variable de départ va être traitée par une ligne de production spécifique.

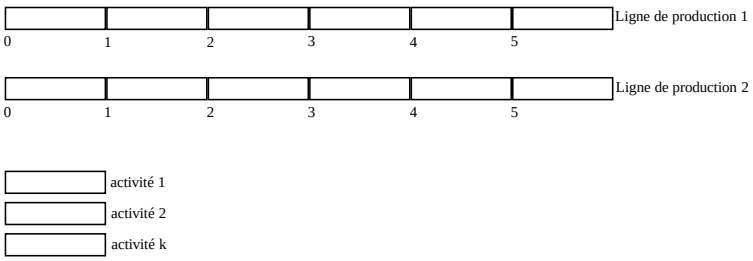

FIGURE 4.4: Open alldifferent.

Le OCSP peut être rapproché de l'ICSP (interactive CSP) [LMM+99, LMMC97]. En effet comme dans le cas de l'OCSP, les domaines sont découverts progressivement grâce a des agents externes. Le filtrage est modifié afin qu'une requête pour découvrir de nouvelles valeurs soit faite des qu'un domaine se vide. Cependant l'ICSP traite des problèmes dont les domaines

peuvent être intégralement découvert grâce aux agents. Ainsi, la résolution d'un ICSP va se concentrer sur l'efficacité des algorithmes de recherche pour résoudre le CSP au lieu de tenter de minimiser le coût des requêtes.

L'avantage des DCSP sur les OCSP est qu'on ne se restreint pas à un ordre monotone, les ajouts et les retraits ne sont soumis à aucune limitation. Ainsi un OCSP peut être vu comme un DCSP dont toutes les modifications se traduisent par une extension des domaines et des contraintes.

4.3.3 Le CSP conditionnel

Le CSP conditionnel proposé dans [MF90] a pour objectif de modéliser des problèmes dont les solutions n'ont pas toutes la même structure (elles ne portent pas toutes sur le même ensemble de variables ou sur les memes contraintes). Cette situation se rencontre fréquemment dans les problèmes de configurations ou les problèmes de conception car le système physique qui doit répondre à un ensemble de contraintes de l'utilisateur n'implique pas les mêmes composants. Ainsi contrairement au DCSP introduit par [DD88] dans lesquels les perturbations touchent le CSP de manière imprévisible, dans les CSP considérés par [MF90] les modifications possibles sont intégrées au modèle. Pour lever cette ambiguïté [SF98] décident de les appeler CSP conditionnel (CCSP). Dans les CCSP les variables sont divisées en deux ensembles : l'ensemble des variables obligatoires et l'ensemble des variables optionnelles. Les contraintes sont aussi réparties en deux ensembles : l'ensemble des contraintes de compatibilité et l'ensemble des contraintes d'activité. Les contraintes de compatibilité sont des contraintes classiques. Les contraintes d'activité définissent les conditions d'activation des variables optionnelles en fonction des variables obligatoires. Une contrainte est activée uniquement si les variables, sur lesquelles elle porte, sont activées. Ainsi durant la résolution d'un CCSP la structure du problème peut évoluer en fonction de l'instantiation courante.

Un CCSP est une restriction d'un DCSP dans lequel toutes les modifications μ possibles sont définies par les contraintes d'activité. On peut écrire un CCSP sous la forme d'un DCSP où CSP_0 est défini par les variables obligatoires et les contraintes de compatibilité et μ correspond à la transformation qui retire les variables optionnelles inactives et qui ajoute les variables optionnelles activées ainsi que les contraintes dont l'ensemble des variables sont activées. Cependant un DCSP présente le gros désavantage d'être binaire (dans un DCSP une variable peut être absente ou présente, mais ne peut pas être potentiellement présente) et peut donc se montrer mon efficace lors de la recherche d'une solution.

Le cas d'étude utilisé dans les deux papiers correspond à un problème de configuration de voiture. Le problème est modélisé par un ensemble de 8 variables correspondant aux différents modules de la voiture : $Package$, $Frame$, $Engine$, $Sunroof$, $AirConditioner$, $Battery$, $Glass$, $Opener$ ainsi

que 12 contraintes d'activité $\{A_1, \ldots, A_{12}\}$ et 6 contraintes de compatibilité $\{C_{13}, \ldots, C_{18}\}$. La variable *Package* a dans son domaine les valeurs *luxury*, *deluxe* et *standard* et la variable *AirConditioner* les valeurs $ac1$ et $ac2$. Une contrainte d'activité correspondra à une contrainte de la forme $Package = luxury \rightarrow Sunroof$. Autrement dit la variable $Sunroof$ sera prise en compte si la variable *Package* a la valeur *luxury*. Un contrainte de compatibilité aura la forme $(Package = standard) \Rightarrow AirConditioner \neq ac2$. Ainsi les contraintes d'activité portent sur les variables (Doivent elles être prises en compte ?) alors que les variables portent sur les valeurs (Telle valeur peut elle être utilisée ?).

Il faut aussi différencier le DCSP des CSP enrichis à l'aide des informations que l'on connaît sur les évolutions possibles du CSP. En effet de nombreuses extensions des CSP exploitent informations disponibles sur les évolutions futures probables : distribution des valeurs, interêt d'une variable, probabilité de présence d'une contrainte dans le CSP final, probabilité qu'une perturbation donnée survienne.

4.3.4 Le MCSP (Mixed Constraint Satisfaction Problem)

Le MCSP proposé dans [FLS96] permet de modéliser des problèmes de décision alors qu'on ne possède qu'une information partielle sur l'état du monde. Dans un MCSP les variables sont réparties en deux groupes. Les variables contrôlables (variables de décisions) sont sous le contrôle d'agents et les variables incontrôlables (variables paramètres) dont la valeur est fixée par l'environnement et sur laquelle l'agent n'a aucun pouvoir. Lorsque toutes les variables sont contrôlables le but de la résolution du CSP est de trouver une décision que l'agent peut mettre en œuvre. Si toutes les variables sont incontrôlables alors le CSP modélise l'état courant du monde. Les auteurs appellent ce type de problème *CSP orienté raisonnement* par opposition au CSP classique qui sont orientés *décision*. En pratique l'utilisateur est souvent confronté à des problèmes hybrides dans lesquels il s'agit de prendre des décisions par rapport à un monde qui n'est pas complètement connu. En fonction de la connaissance que l'agent a du monde les solutions recherchées ne répondent pas à la même question. Lorsque le monde est *totalement observable* c-à-d que l'agent connaitra complètement l'état du monde avant de mettre la solution en application on va chercher des solutions qui pourront s'appliquer aux différents états pouvant survenir et parmi lesquelles l'agent n'aura plus qu'a choisir. Lorsque le monde n'est *pas observable* toutes la connaissance, dont disposera l'agent lors de l'utilisation de la solution, est contenue dans le CSP une solution intéressante sera une décision (une instantiation des variables de décisions) consistante quelque soit l'état du monde (instantiation des variables d'état).

Un exemple de MCSP pourrait être un CSP correspondant à un problème prenant en compte la température extérieure. La température serait modélisée par une variable dans le domaine correspondrait à l'ensemble des températures possibles et sur laquelle l'utilisateur n'aurait aucune influence. Ainsi comme expliqué plus haut, l'utilisateur pourrait soit rechercher un solution robuste, soit rechercher un ensemble de solutions correspondant à l'ensemble des situations possibles.

Définition 7. Un MCSP est défini par un tuple $\mathcal{P} = \langle \Lambda, L, X, D, \kappa, \mathcal{C} \rangle$ où :
- $\Lambda = \{\lambda_1, \ldots, \lambda_p\}$ est un ensemble de paramètres ;
- $L = L_1 \times \cdots \times L_p$, où L_i est le domaine λ_i ;
- $X = \{x_1, \ldots, x_n\}$ est un ensemble de variables de décision ;
- $D = D_1 \times \cdots \times D_n$, où D_i est le domaine de x_i ;
- κ est l'ensemble des contraintes portant seulement sur des paramètres ;
- $\mathcal{C}$ est un ensemble de contraintes, chacune portant sur au moins une variable de décision.

On peut définir un MCSP sous la forme d'un DCSP avec $CSP_0 = \langle X, D, \mathcal{C}' \rangle$, où $\mathcal{C}'$ correspond à l'ensemble des contraintes ne portant que sur des variables de décisions, et μ défini par (Λ, L, κ) afin de prendre en compte les transformations impactant les paramètres.

4.3.5 Le SCSP (Stochastic Constraint Satisfaction Problem)

Le SCSP proposé dans [Wal02] permet de modéliser les problèmes possédant des incertitudes sur le passé (du fait des approximations lors des mesures) et sur le futur (qui ne peut pas être prédit exactement). Comme pour le MCSP, les variables sont séparées en deux groupes : les variables contrôlables (variables de décisions) et les variables incontrôlables (les variables stochastiques). Mais à la différence des MCSP, dans les SCSP une probabilité de distribution est associée au domaine de chaque variable stochastique. Une requête peut être de construire une décisions (une instantiation des variables de décision) qui maximise la probabilité de consistance dans le monde réel. Ainsi comme pour les MCSP, un SCSP peut s'exprimer sous la forme d'un DCSP.

L'exemple utilisé pour illustré le SCSP correspond à un problème de planification de la production de livres durant m quart d'heure. Durant chaque quart d'heure il y a une chance équivalente de vendre entre 100 et 105 copies d'un livre. Pour que les consommateurs soient satisfaits on tente de satisfaire la demande à 80% chaque quart d'heure. Au début de chaque quart d'heure on décide combien de livres on souhaite imprimer pour le quart d'heure en cours. Ce problème est modélisé à l'aide d'un SCSP à m stages. Il y a m variables de décisions x_i représentant la production de livre pour le i^{eme} quart d'heure. Il y a m variables stochastiques y_i représentant la demande durant le i^{eme} quart d'heure et prenant une valeur entre 100 et 105 avec la même probabilité. On

veut donc que durant le premier quart d'heure tous les clients soient satisfaits. On veut que durant le second quart d'heure tous les clients soient satisfaits en prenant en compte les demandes insatisfaites ou les stocks du premier quart d'heure $x_1 \geq y_1$ et que $x_2 \geq y_2 + (y_1 - x_1)$. En généralisant, pour le j^{eme} quart d'heure on souhaite que $x_j \geq y_j + \sum_{i=1}^{j-1}(y_i - x_i)$. On doit satisfaire chacune de ces m contraintes avec une probabilité de 0,8. La solution est simple puisqu'il suffit d'en produire le maximum possible à chaque fois. Cependant si on ajoute au problème des coûts de stockage le problème se complique.

4.3.6 Le BCSP (Branching Constraint Satisfaction Problem)

Le BCSP proposé dans [FB00b, FB00a] modélise des problèmes de décision qui reçoivent périodiquement des ajouts. Les auteurs prennent comme exemple un cas de planification de tâches dans lequel on sait que des tâches vont être ajoutées régulièrement aux problèmes sans savoir lesquelles exactement. Le but est donc de retenir les planifications qui pauseront le moins de problème avec les potentielles évolutions du problème. A chaque étape, les variables présentes sont affectées en prenant en compte les variables qui vont être ajoutées au problème. A chaque variable est associée une utilité. Ainsi, à chaque étape une probabilité d'ajout peut être calculée pour chaque variable qui ne fait pas encore partie du problème. Le but est de fixer les variables du problème à des valeurs qui maximisent l'utilité globale c-à-d qui permet l'affectation du maximum de variables possibles sans générer de conflits.

On considère un problème d'emploi du temps dans lequel de nouvelles tâches apparaissent. Le but est de planifier le plus de tâches (définies par leur durée, leur consommation et leur utilité) possibles avant leur date de réalisation. Il y a deux ressources identiques que les tâches peuvent utiliser. Initialement deux tâches sont présentes, la tâche A et la tâche B. On sait qu'une nouvelle tâche va apparaitre au temps 1. Il y a une probabilité de $0,6$ que ce soit la tâche C et une probabilité de $0,4$ que ce soit la tâche D. Les tâches A et B doivent être planifier avant de connaitre la troisième tâche. Le but de BCSP est de permettre de répondre à la question : Comment les tâches A et B doivent elles être planifiées.

Contrairement au DCSP, le BCSP intègre un modèle des évolutions qui vont survenir. On peut exprimer un BCSP sous la forme d'un DCSP où CSP_0 correspond aux taches du CSP initial et μ correspond à l'ajout des differentes taches à chaque étape.

Les MCSP, PCSP, SCSP ou BCSP peuvent être qualifiés de CSP riches puisque l'utilisateur fournit des informations supplémentaires au sujet de la nature ou de la probabilité des perturbations qui peuvent survenir. Le fait que les perturbations soient déjà incluses dans le modèle permet de chercher des solutions a priori robustes avant de connaître les valeurs effectivement

utilisées. Ils permettent aussi de trouver, a posteriori, des solutions prenant en compte les évolutions de l'environnement.

A l'inverse le DCSP est un cadre pauvre mais très général puisqu'il ne fait aucune supposition sur les modifications qui vont survenir. Contrairement aux autres cadres, le DCSP ne prend pas en compte les perturbations dans le modèle et n'utilise pas d'information sur la probabilité des événements futurs. Lorsqu'une perturbation survient le DCSP adapte le CSP utilisé afin de refléter le nouvel état du problème et tente de le résoudre. Le DCSP est donc extrêmement souple, en contre-partie les informations disponibles pour faire face aux perturbations sont limitées puisque elles ne peuvent être issues uniquement des tentatives de résolutions des CSP précédents.

4.4 Méthodes de résolution du DCSP

Comme nous l'avons vu dans la partie 4.2, un DCSP est une séquence de CSP statiques pour laquelle on souhaite trouver une solution. Pour trouver une solution on pourrait se concentrer sur chaque CSP et tenter de les résoudre independamment les uns des autres à l'aide des méthodes de résolution classiques. Ce serait passer passer a cote de la spécificité des DCSP : le caractère incrémental de la séquence de CSP. De plus il ne faut pas oublier qu'un DCSP s'inscrit dans le temps : souvent lorsque les perturbations surviennent, un partie de la solution à déjà été réalisé. On ne peut donc pas résoudre en dehors de son contexte. En pratique, on se limite généralement au CSP précédent. Il existe plusieurs façons d'exploiter les informations de la recherche précédente.

La question posée par la résolution d'un DCSP est donc comment calculer une solution $\mathcal{S}'$ d'un problème $\mathcal{P}'$ alors que l'on connait la solution $\mathcal{S}$ du problème $\mathcal{P}$ tel que $\mathcal{P}' = \mu(\mathcal{P})$.

[SV95] classent les approches en deux grands type : les approches où l'information mémorisée et réutilisée portent sur les solutions, celles dans lesquelles elles portent sur les informations decouvertes durant le filtrage et celles portant sur les contraintes.

On peut identifier plusieurs approches :

- Les approches par réutilisation de solutions utilisent $\mathcal{S}$ pour calculer $\mathcal{S}'$
- Les approches par réutilisation de raisonnements se concentrent sur μ (en différenciant les cas correspondant à des restrictions des cas correspondant à des relaxations) pour adapter $\mathcal{S}$
- Les approches par réutilisation de contraintes peuvent être vues comme des approches sémantiques adaptant $\mathcal{P}$ en modifiant les contraintes.

4.4.1 Les approches par réutilisation de solution

Le premier type d'approche repose sur l'idée que si une solution $\mathcal{S}$ a été trouvée pour un problème $\mathcal{P}$ et que le problème $\mathcal{P}'$ est proche du problème $\mathcal{P}$ alors une solution $\mathcal{S}'$ de $\mathcal{P}'$ doit se trouver dans le voisinage de $\mathcal{S}$.

Ces approches supposent que la seule information transmise au problème $\mathcal{P}_{n+1}$ est une solution du problème $\mathcal{P}_n$ [MJPL90] [MJPL92]. Les approches reposant sur la réutilisation des solutions s'appliquent dans les situations où le problème précédent, $\mathcal{P}_n$, a été résolu ou prouvé inconsistant, une solution $\mathcal{S}_n$ de $\mathcal{P}_n$ a été trouvée et enregistrée, une modification est survenue dans la définition du problème $\mathcal{P}_n$ créant le problème $\mathcal{P}_{n+1}$. Si $\mathcal{P}_{n+1}$ est une relaxation de $\mathcal{P}_n$ alors $\mathcal{S}_n$ est aussi une solution de $\mathcal{P}_{n+1}$. Si la modification est une restriction du problème la solution $\mathcal{S}_n$ peut ne plus être une solution. L'hypothèse sur laquelle repose l'approche par réutilisation de solution est : puisque que $\mathcal{P}_{n+1}$ est proche de $\mathcal{P}_n$ alors une solution de $\mathcal{P}_{n+1}$ peut raisonnablement être cherchée dans le voisinage de $\mathcal{S}_n$. Si cette hypothèse est correcte, l'approche présente le double avantage d'être efficace et stable. En fonction des situations l'hypothèse peut ne pas se vérifier et l'approche être totalement inefficace.

On peut distinguer quatre approches reposant sur ce principe :

La *recherche arborescente* : le plus simple pour réutiliser une solution existante, dans le contexte d'une recherche arborescente en profondeur d'abord (DFS), est de l'utiliser comme heuristique [HP91]. Ainsi dans la nouvelle recherche, pour chaque variable, on teste en priorité la valeur qui lui est assignée dans la solution. Pour prolonger ce principe (explorer des instantiations proche de la solution initiale) il peut être intéressant d'utiliser un LDS (limited discrepancy search) en augmentant peut à peu la distance autorisée par rapport à la solution plutôt qu'un DFS [HG95].

La recherche locale s'adapte bien au contexte dynamique. La solution précédente est utilisée comme point de départ pour la recherche locale. Ensuite à chaque étape la recherche locale va réparer l'instantiation courante pour la rendre consistante et explorer son voisinage. Par exemple, [MJPL92], construit un voisinage basé sur le nombre d'affectations en communs. L'avantage de ce type d'approche est qu'il permet généralement d'assurer une certaine stabilité à l'ensemble des solutions trouvées.

La méthode par perturbation de solution . Le principe est de préparer à l'avance un ensemble de méthodes qui pourront être appliquées rapidement si une modification survient. Initialement, ces méthodes se limitent aux réseaux de contraintes acycliques qui impliquent uniquement des contraintes qui utilisent un ensemble de variables d'entrée pour déterminer une unique variable de sortie. Des extensions ont été proposées pour gérer les réseaux de contraintes cycliques ainsi que d'autres type de contraintes [FBMB90].

La méthode par relaxation-affectation de variable proposée dans [Sch94] consiste à modifier la solution précédente en une nouvelle solution réalisable pour le problème. Le principe de cette méthode est d'identifier les

contraintes insatisfaites et de relaxer pour chacune d'elles au moins une variable, puis de leur chercher une nouvelle affectation. Et recommencer de manière itérative jusqu'à ce que toutes les contraintes soient satisfaites. La différence avec une méthode de recherche locale et qu'elle est complète lorsqu'elle est effectuée dans un arbre de recherche. Le fait qu'elle repose sur les contraintes pour la phase d'exploration du voisinage, permet à cette methode d'etre efficace dans les espaces de recherche contraints. Alors qu'une méthode de recherche locale classique perdrait beaucoup de temps à trouver une affectation réalisable, l'utilisation des contraintes permet d'eliminer rapidement certaines affectations inconsistantes.

Le LNS peut être vue comme une combinaison des méthodes arborescentes et des méthodes par relaxation-affectation. En effet avec LNS il s'agit de relaxer un sous ensemble des variables de la solution initiale. De plus la recherche réalisée sur le sous ensemble de variable peut s'appuyer sur un LDS [HG95]. Le LNS va plus loin puisqu'en plus de rechercher une solution realisable en utilisant le principe de relaxation-affectation, il tente d'exploiter la structure du probleme pour identifier les affectations problematiques. De plus pour garantir un plus grande stabilité, on peut appuyer la recherche sur un LDS. LNS semble donc etre une methode interessante, nous l'aborderons plus longuement dans la partie ? .

4.4.2 Les approches par réutilisation du raisonnement

Ces approches reposent sur la réutilisation des contraintes découvertes durant la résolution du problème $\mathcal{P}_n$ pour la résolution du problème $\mathcal{P}_{n+1}$ [HP91]. Les contraintes sont produites durant la recherche. En effet, la propagation de l'ensemble des contraintes, permet d'identifier des affectations inconsistantes. Ce sont ces affectations qui sont stockées sous la forme de tuples interdits afin de faciliter les recherches suivantes. Les méthodes utilisées pour produire ces contraintes et la façon dont ses tuples sont stockés est détaillés dans la partie 6.

Ses approches gèrent les situations où le problème précédent $\mathcal{P}_n$ a été résolu et prouvé consistant ou inconsistant, la résolution a permis la génération d'un ensemble $\mathcal{I}_n$ de contraintes (conséquence des contraintes de $\mathcal{P}_n$). Puis une perturbation est survenu dans la définition de $\mathcal{P}_n$ et un problème $\mathcal{P}_{n+1}$ a été produit. En cas de relaxation, les contraintes de $\mathcal{I}_n$ ne sont plus forcément valides. Si $\mathcal{P}_{n+1}$ est une restriction de $\mathcal{P}_n$, alors $\mathcal{I}_n$ est encore valide dans $\mathcal{P}_{n+1}$. L'hypothèse des approches par réutilisation du raisonnement est : puisque $\mathcal{P}_n$ et $\mathcal{P}_{n+1}$ sont proches alors la majorité des contraintes de $\mathcal{I}$ doivent être encore valides. Si l'hypothèse est vérifiée, le fait de stocker celles qui sont encore valides permet de les réutiliser pour éviter de revisiter une zone de l'espace de recherche qu'on sait ne contenir aucune solution. Ainsi, l'efficacité de ces méthodes repose sur la capacité à retirer de $\mathcal{I}_n$ les contraintes qui ne sont

plus valides ($\mathcal{I}'_n$) afin de générer un ensemble de contraintes $\mathcal{I}_{n+1}$ valides pour $\mathcal{P}_{n+1}$. La différence entre les approches repose sur les informations qui sont utilisées pour déterminer l'ensemble $\mathcal{I}'_n$ après chaque perturbation.

- Les méthodes basées sur *les graphes* utilisent uniquement le graphe du réseau de contraintes pour déterminer si une contrainte doit être remise en question ou pas. Ainsi, si une contrainte c est retirée alors, toutes les contraintes produites en utilisant c doivent être remises en question et peuvent être retirées à leur tour.
- Les méthodes basées sur *les justifications* utilisent une justification qui a été enregistrée avec chaque contrainte induite c' pour déterminer si c' doit être remise en question. La justification d'une contrainte c' est l'ensemble C des contraintes ayant servi à la générer (contraintes initiales et contraintes induites). On prend l'exemple d'un réseau de contraintes binaires portant sur 4 variables (Figure 4.5). Le solveur décide ensuite de fixer la valeur de la variable v_2 à 2. Le retrait des valeurs 1 et 3 du domaine de var_2 entraine la suppression de valeurs dans les domaines des autres variables par filtrage et propagation. A chaque retrait est associé une explication rendant compte des causes qui ont mené à la suppression de la valeur (Figure 4.6). Le solveur fixe ensuite la variable var_1 à la valeur 1. Après la phase de propagation, une solution a été découverte (Figure 4.7).
- Les méthodes basées sur *les explications* différent des méthodes basées sur les justifications par la nature des informations stockées avec chaque contrainte induite c'. L'information enregistrée est une explication de c' c'est à dire un ensemble C de contraintes initiales qui sont responsables de sa création. Ainsi lorsqu'une contrainte initiale c est retirée l'ensemble des contraintes dont l'explication contient c peuvent être retirées en une passe. On reprend le même exemple que précédemment (Figures 4.5, 4.8 et 4.9). On observe la différence entre les justifications et les explications. Alors que les justifications ne prennent en compte que la contrainte impliquée directement lors du retrait, les explications remontent jusqu'aux décisions responsables. Ainsi on peut calculer récursivement une explication à partir des justifications.

Cette approche [SV94], proposée dans [Dec90], repose sur l'apprentissage, durant la résolution d'un CSP par un algorithme de backtrack, d'instantiations qui ne sont pas globalement inconsistantes. Ces instantiations définissent des contraintes qui peuvent être utilisées dans plusieurs contextes : pour le maintien des solutions durant une restriction, ainsi que lors de la relaxation en utilisant des algorithmes adaptés. Ainsi au lieu de stocker uniquement les instantiations inconsistantes, on construit des nogoods définis par des paires (A, C') où A est l'instantiation et C' un sous ensemble de C (C l'ensemble des contraintes du CSP) tel que l'assignation A soit inconsistante si on restreint C à C'.

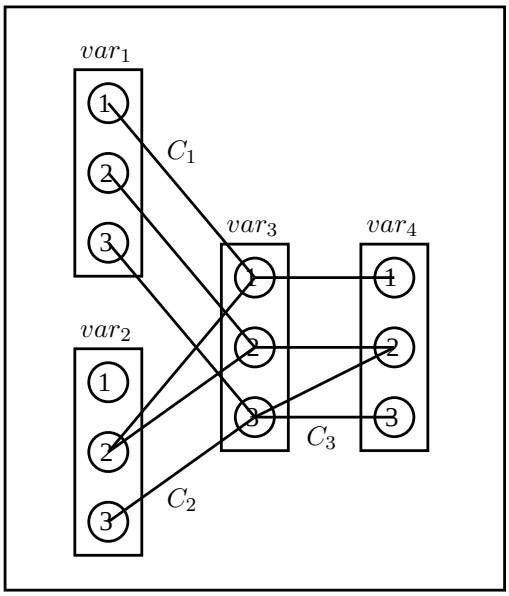

FIGURE 4.5: On considère un réseau de contraintes binaires portant sur 4 variables. Les traits représentent les supports entre les différentes valeurs.

4.4.2.1 Présentation des explications

L'utilisation des *nogoods* [SV94] permet aux méthodes rétrospectives de détecter précisément les affectation responsables de l'échec d'une affectation partielle. Ainsi on exploite les résultats sur les anciennes branches pour améliorer le parcours des branches non encore visitées et donc pour adapter l'exploration de l'espace de recherche à la structure du problème.

Définition 8. Pour un CSP (V,D,C), un nogood est une paire (A,J), où A est une affectation (totale ou partielle) et J un sous-ensemble de C tel que aucune solution du CSP ne contienne A sans violer J.

On considère un CSP représenté par un couple (V, C) où V est un ensemble de variables et C un ensemble de contraintes portant sur ces variables. Les domaines peuvent être considérés comme des contraintes unaires. De plus le mécanisme d'énumération est géré par une série d'ajouts et de retraits de contrainte. Ces contraintes sont appelées *contraintes de décision*. Une explication de contradiction (aussi appelé *nogood*) est un sous ensemble des contraintes actuelles du problème qui, seules, mènent a une contradiction (aucune solution ne contient un *nogood*). Une explication de contradiction se divise en deux parties : un sous-ensemble de l'ensemble original des contraintes ($C' \subset C$) et un sous-ensemble des contraintes de décision introduites jusque là dans la recherche ($dc_1, \ldots, dc_k$). Explication de contradiction :

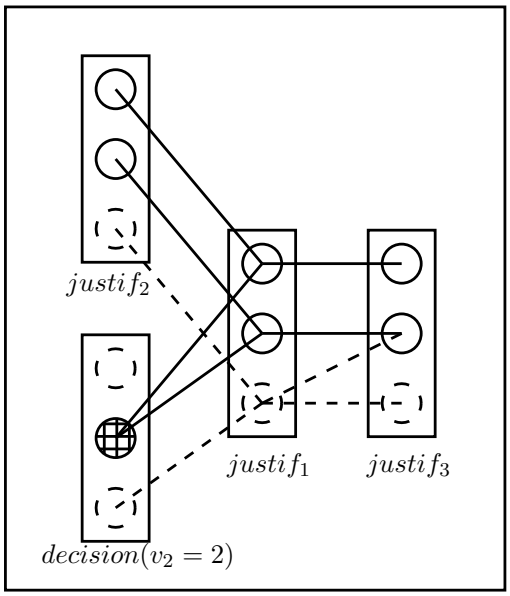

FIGURE 4.6: Le solveur fixe la valeur de la variable var_2 à 2. Les valeurs 1 et 2 sont alors supprimées du domaine de var_2. Ainsi la valeur 3 de var_3 perd son unique support et est à son tour supprimé ($just_1 = (var_2 \neq 3) \wedge C_2$). Les valeurs 3 de var_1 et var_4 perdent alors aussi leur support ($just_2 = (var_3 \neq 3) \wedge C_1$ et $just_3 = (var_3 \neq 3) \wedge C_3$).

$$\neg(C' \wedge dc_1 \wedge \cdots \wedge dc_k)$$

Une explication de contradiction peut être réécrite en une explication de retrait :

$$C' \wedge (\bigwedge_{i \in [1,k] \setminus j} dc_i) \rightarrow \neg dc_j$$

Considérons la contrainte de décision j : $(v = a)$ dans la formule précédente. La partie gauche de l'implication est appelé explication de retrait parce qu'il justifie le retrait de la valeur a du domaine de la variable v. On le note : $expl(v \neq a)$.

Les opérations de filtrage dans un CSP peuvent être considérées comme une séquence de retrait de valeurs qui peuvent être expliqués. La plus simple des explications est de considérer l'ensemble de toutes les contraintes actives (l'ensemble des contraintes initiales et l'ensemble des contraintes de décision). Cependant une explication portant sur une grand nombre de contraintes permet d'apprendre peu de choses. On essai donc de construire des explications aussi précises que possibles.

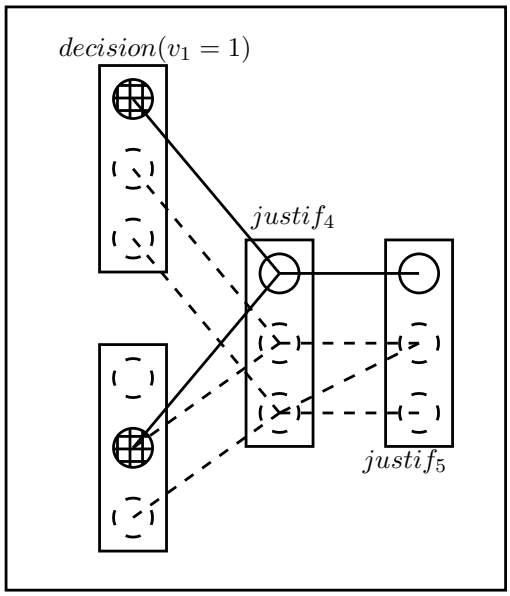

FIGURE 4.7: Le solveur prend alors la decision de fixer var_1 à 1. Ce qui entraine le retrait de la valeur 2 de var_3 ($just_4 = (var_1 \neq 2) \wedge C_1$). Puis le retrait de 2 du domaine de var_4 ($(just_5 = (var_3 \neq 2) \wedge (var_3 \neq 3) \wedge C_3)$).

Les explications peuvent être combinées entre elles pour en produire de nouvelles. En effet supposons que $dc_1 \vee \cdots \vee dc_j$ est l'ensemble des choix possibles pour une décision donnée. S'il existe un ensemble d'explications $C'_1 \rightarrow \neg dc_1, \ldots, C'_j \rightarrow \neg dc_j$ une nouvelle explication peut être dérivée : $\neg(C'_1 \wedge \cdots \wedge C'_j)$. De plus, à partir du domaine vide d'une variable v, on peut calculer une explication de contradiction :

$$\neg(\bigwedge_{a \in dom(v)} expl(v \neq a))$$

Généralement il existe plusieurs explications d'élimination pour le retrait d'une valeur. Enregistrer toutes ces explications mène à une complexité spatiale exponentielle.

Les explications peuvent être utiliser dans trois buts différents.

Explications utilisateurs : Les explications peuvent être utilisées pour aider l'utilisateur à comprendre l'interaction entre les différentes contraintes. Par exemple, les explications permettent au solveur d'envoyer à l'utilisateur un message expliquant pourquoi le domaine d'une variable a été totalement vidé et pourquoi il n'y a pas de solution au problème posé. Ou encore expliquer pourquoi une variable ne peut pas prendre une valeur a.

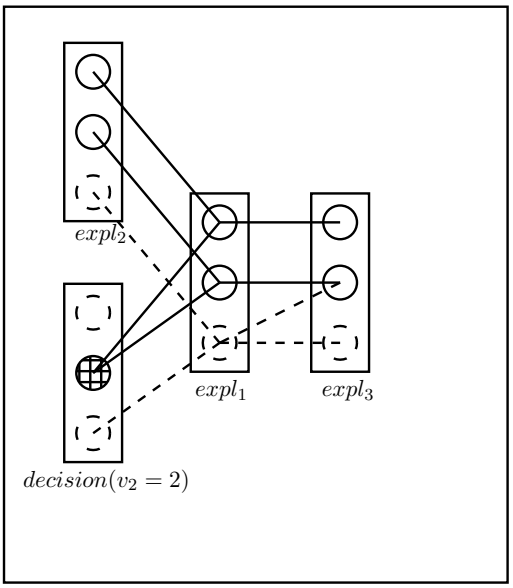

FIGURE 4.8: Le solveur fixe la valeur de la variable var_2 à 2. Les valeurs 1 et 2 sont alors supprimées du domaine de var_2. Ainsi la valeur 3 de var_3 perd son unique support et est à son tour supprimé ($expl_1 = decision(var_2 = 3) \wedge C_2$). Les valeurs 3 de var_1 et var_4 perdent alors aussi leur support ($expl_2 = decision(var_2 = 3) \wedge C_2 \wedge C_1$ et $expl_3 = decision(var_2 = 3) \wedge C_2 \wedge C_3$).

Resolution des problèmes surcontraints : Les explications permettent d'identifier le sous ensemble de contraintes en conflits. L'utilisateur peut alors choisir, en fonction de ses préférences, quelle contrainte il préfère supprimer du problème.

Retrait dynamique de contraintes : Les explications peuvent être utiles pour gérer des problèmes dynamiques. Ajouter incrémentalement une contrainte à un problème est un sujet qui a largement été étudié mais le retrait incremental d'une contrainte n'est pas aussi évident. [Bes91] utilisait déjà un système d'explication simplifié (justifications) pour permettre le retrait. Dans un système basé sur les explications le processus de retrait peut être réalisé de la manière suivante :

1. déconnecter la contrainte du réseau de propagation
2. restaurer les valeurs dont la suppression était du (de manière direct ou indirect) à la contrainte retirée. Pour cela il s'agit de prendre en compte les événements associés à des explications qui contiennent la contrainte retirée
3. vérifier que les valeurs réintroduites sont consistantes avec les contraintes restantes

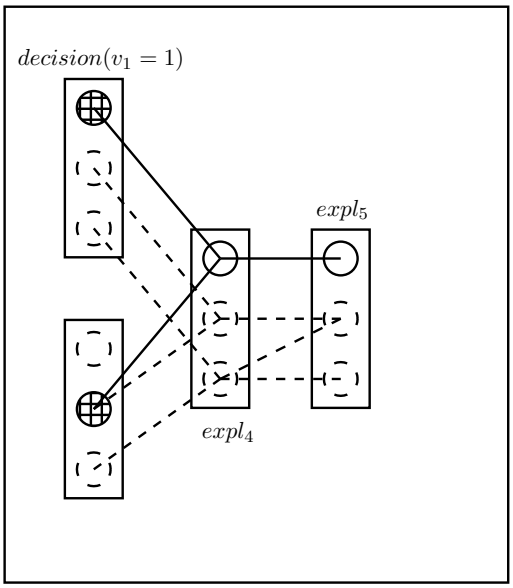

FIGURE 4.9: Le solveur prend alors la décision de fixer var_1 à 1. Ce qui entraine le retrait de la valeur 2 de var_3 ($expl_4 = decision(var_1 = 1) \wedge C_1$). Puis le retrait de 2 du domaine de var_4 ($(expl_5 = decision(var_1 = 1) \wedge C_1 \wedge decision(var_2 = 3) \wedge C_2 \wedge C_3)$).

4. propager les éventuels retrait de valeur

A la fin du processus le système est dans un état consistant : un état qui aurait été atteint si la contrainte retirée n'avait jamais été introduite dans le système.

4.4.3 Les approches par réutilisation des contraintes

En plus de ces méthodes reposant sur l'utilisation des informations portant sur l'implication de chaque contrainte dans le filtrage, il existe des méthodes exploitant l'état interne des contraintes.

Les algorithmes de filtrages permettant d'obtenir l'arc consistance sont les plus courants. Malheureusement les algorithmes d'arc-consistance sont prévus pour un contexte statique, ils ne peuvent donc pas être utilisés directement dans le cadre des DCSP à moins de devoir répéter le même travail. Par exemple, l'algorithme AC4 permet facilement de prendre en compte une restriction mais lors d'une relaxation les raisons qui ont mené au retrait d'une valeur ont été oublié. On ne peut donc pas faire le travail inverse. Ainsi dans [Bes91] l'auteur décrit un algorithme, DnAC4, pour maintenir l'arc consistance dans les DCSP. DnAC4 est une extension de l'AC4. En plus de stocker l'information sur le nombre de supports disponible, DnAC4 stocke des informations durant le filtrage qui lui permettent d'être incrémental lors des

relaxations. Ainsi à chaque fois qu'une valeur est supprimée, le DnAC4 enregistre la contrainte qui est à l'origine de son retrait comme justification. La justification est la première contrainte pour laquelle la valeur n'a plus de support. Lors d'une relaxation, grâce aux justifications, on peut réintégrer dans les domaines les valeurs dont le retrait avait été causé par la contrainte.

DnAC4 initialement limité aux CSP binaires a ensuite été étendu au cas général [Bes92]. Cette fois ci l'algorithme dynamique s'inspire de l'algorithme de filtrage de consistance d'arc global GAC4 pour développer l'algorithme DnGAC4. Ainsi pour chaque valeur retirée l'algorithme DnGAC4 enregistre une justification et pour chaque tuple retiré il garde la valeur responsable de son retrait (killer). Le killer est la première valeur retirée du tuple. Afin de réduire l'espace [Deb96] reprend le principe de DnAC4 et l'applique à l'algorithme AC6. Dans [NB94] les auteurs préfèrent privilégier la simplicité de la méthode à son efficacité et proposent une approche, appelée AC—DC, ne nécessitant pas de stocker des informations complémentaires sur l'état passé du CSP. Ainsi dans le cadre des CSP binaires la méthode consiste à déterminer un sur-ensemble de l'ensemble des valeurs à réintroduire dans le domaine des variables u et v lors du retrait d'une contrainte portant sur (u, v). En fonction de la quantité d'information stockée la complexité en espace est plus ou moins élevée. En contrepartie les algorithmes stockant peu d'information possèdent une complexité en temps plus importante à cause du surcout engendré par le calcul de l'information manquante 4.10. Toutes ces méthodes reposent sur des algorithmes génériques d'arc-consistance et se concentrent sur le cas des CSP binaires. Elles ne conviennent donc pas aux CSP comportant des contraintes globales.

	DnAC4	DnAC6	AC—DC
Complexité en espace	$O(ed^2 + nd)$	$O(ed + nd)$	$O(e + nd)$
Complexité en temps	$O(ed^2)$	$O(ed^2)$	$O(ed^3)$

FIGURE 4.10: Complexité des algorithmes dynamiques de consistance dans le cas d'un CSP à e contraintes binaires, n variables et dont le plus grand domaine possède d valeurs.

En s'attaquant efficacement à des sous-problemes récurrents, les contraintes globales deviennent un élément inévitable dans un CSP. Il peut donc être intéressant de tenter d'étendre le principe de la réutilisation de contrainte dans le cadre des contraintes globales. Bartak présente plusieurs propositions dans [Bar03]. En effet d'un point de vue sémantique, ajouter une variable à un CSP implique de pouvoir faire prendre en compte cette variable par les contraintes. Ainsi au lieu de retirer puis reposer chaque contrainte, une autre approche serait de modifier l'état interne de la contrainte pour prendre l'ajout sans avoir à passer par le retrait effectif de la contrainte. Nous reviendrons sur ces questions dans la partie 7.

4.5 Conclusion

Les différents formalismes présentés intègrent aux modèles la dimension dynamique du problème. Au contraire le DCSP ne fait aucune supposition sur les évolutions futures. Ce sont les méthodes de recherche utilisées pour traiter les DCSP qui permettent d'adapter le modèle lors des modifications. Il existe 3 grandes familles d'approches : celles basées sur la réutilisation de solutions, celles basées sur la réutilisation de raisonnement et celles basées sur la réutilisation de contraintes. Dans les chapitres suivant nous allons explorer chacune des approches au travers de la méthode Large Neighborhood Search, de l'utilisation des nogoods et de la monotonie et incrémentalité des contraintes.

Chapitre 5

Large Neighborhood Search

La recherche locale est une technique utilisée en pratique pour aborder les problèmes d'optimisation combinatoire difficiles. L'idée est d'améliorer de façon itérative une solution existante en explorant un voisinage de celle-ci [LK73]. Les solutions dites voisines sont généralement obtenues en appliquant des transformation (aussi appelées mouvements) plus ou moins importantes à la solution courante. Étant donnée une fonction objective f, la résolution d'un problème de satisfaction de contraintes, à l'aide de la recherche locale, se déroule en 4 temps :

1. Générer une solution T satisfaisant l'ensemble des contraintes
2. Chercher une solution T' en transformant T
3. Si $f(T) > f(T')$, remplacer T par T' et recommencer l'étape 2
4. Si aucune meilleure solution ne peut être trouvée, alors T est un optimum local. On recommence alors depuis l'étape 1, jusqu'à obtenir une solution satisfaisante ou dépasser la limite de temps.

Ainsi une grande part de l'efficacité d'une recherche locale est du à l'ensemble des transformations autorisées. De plus, le fait de partir d'une solution faisable pour explorer l'espace de recherche permet d'assurer une certaine stabilité dans les solutions proposées. Nous nous sommes interessés au cas de la méthode du Large Neighbourhood Search (LNS) qui permet de combiner la recherche locale et la programmation par contraintes.

5.1 Large Neighbourhood Search

La méthode du Large Neighbourhood Search (LNS) a été introduite par Paul Shaw dans [Sha98, KPS98a] afin d'utiliser la programmation par contraintes pour résoudre des problèmes dont la taille est trop importante pour permettre une approche contrainte classique. Un point essentiel, lors de la mise en place d'une recherche locale, est la définition du voisinage et des mouvements autorisés durant l'exploration. La définition du voisinage est souvent spécifique à un problème et est d'autant plus difficile à définir que le nombre de contraintes additionnelles est élevé. En effet, au problème de trouver des voisins intéressants s'ajoute celui de trouver des voisins consistants. L'intérêt d'utiliser la programmation par contraintes dans le cadre d'une recherche locale est de définir de façon générique la notion de voisinage même lorsque le problème possède de nombreuses contraintes additionnelles [KPS+98b]. La recherche locale s'adapte généralement mal à la recherche arborescente.

Ainsi, en s'inspirant des méthodes de *shuffling* utilisées pour l'ordonnancement des tâches, Paul Shaw met au point le LNS. Le LNS fonctionne sur le principe de la relaxation - optimisation. Par exemple, dans le cadre du `VRP`, à chaque étape un sous-ensemble des visites est retiré du plan de route solution et est réoptimisé. La réoptimisation profite de tout les raisonnement utilisables en programmation par contraintes : filtrage, heuristiques, ... Chaque itération

peut être vue comme une exploration du voisinage de la solution. Ainsi, les deux éléments clefs de cette méthode sont : la sélection du sous-ensemble de variables à relaxer et la méthode utilisée pour compléter les solutions partielles.

5.1.1 Description générale du LNS

LNS est une technique générique de recherche locale qui fonctionne sur le principe *relaxation - reoptimisation*, c'est-à-dire en relaxant des sous-parties de la solution, puis en réoptimisant les sous-parties relaxées afin de trouver des solutions de coût plus faible. LNS nécessite donc de pouvoir détecter les variables qu'il est intéressant de relaxer en même temps et de disposer de méthodes pour compléter une solution partielle du problème qu'on souhaite résoudre. La méthode de construction utilisée est typiquement une méthode heuristique ou une méthode basée sur une recherche arborescente.

L'exemple d'un problème de coloration de graphe est donné dans [KPS98a]. La méthode de construction de solutions consiste à affecter une couleur à chaque nœud (sans prendre en compte les contraintes de différences entre nœuds adjacents). La fonction de coût renvoie le nombre de nœuds adjacents ayant la même couleur. Trouver une solution au problème de coloration signifie alors d'avoir un coût de zéro. LNS procède donc en choisissant un sous-ensemble de nœuds qu'elle relaxe (elle désaffecte la couleur qui leur avait été donnée). Une recherche arborescente est ensuite utilisée pour réaffecter une couleur à chaque nœud du sous-ensemble en minimisant le nombre de nœuds adjacents ayant la même couleur.

L'algorithme général du LNS est donné Algorithme 3 [KPS98a]. Il prend une solution s^* et cherche à l'améliorer par *relaxation - reoptimisation*. Le nombre initial de décisions à relaxer est passé en paramètre.

Algorithm 3 LNS(s^*, *size*, *randomness*)

```
1: while ¬ StopNow do
2:    s := RelaxSolution(s*, size, randomness);
3:    s := Rebuild(s);
4:    if cost(s) ≤ cost(s*) then
5:       s* := s;
6:    end if
7:    size := AdjustSize(size);
8: end while
9: return s*
```

StopNow La fonction `StopNow` définit la condition d'arrêt de LNS. La taille du voisinage explorable est $C^{size}_{|s^*|}$, il est donc souvent impossible pour la méthode de recherche de vérifier qu'on a effectivement trouvé un minimum. Il

faut donc définir une condition d'arrêt. Généralement la condition d'arrêt est basée sur le temps et correspond au délai que l'utilisateur est prêt à attendre pour obtenir une solution. Dans un environnement interactif, il peut ne pas avoir de condition d'arrêt, le solveur renvoie, à l'utilisateur, chaque nouvelle amélioration trouvée. L'utilisateur décide ensuite si la nouvelle solution lui convient ou si il souhaite que le solveur continue son exploration.

RelaxSolution La fonction `RelaxSolution` peut aller du plus simple, décider de relaxer r décisions choisies aléatoirement, au plus complexe, relaxer des décisions qu'on aura détecte comme interdépendantes. En effet, si les variables relaxées ne sont pas interdépendantes il n'y a pas de raisons de les relaxer en même temps puisque chacun des traitements aurait pu être effectuées de manière indépendante.

Le paramètre *randomness* varie entre 0 et 1. Il indique la proportion de variables choisies aléatoirement et celle de variables choisies grâce aux relations d'interdépendance. Ce paramètre permet de diversifier la recherche et d'éviter la stagnation dans des optima locaux.

Rebuild La fonction `Rebuild` prend une solution partiellement relaxée et la complète en utilisant une approche constructive. N'importe quelle méthode peut être utilisée durant cette phase. Cependant, comme les approches heuristiques peuvent prendre des décisions incorrectes durant la reconstruction, une recherche arborescente semble être une des méthodes les plus intéressantes. Il peut s'agir d'une méthode complète basée sur un *branch and bound*, ou d'une exploration partielle de l'arbre basée sur des méthodes comme le `LDS` [HG95] (cf. section 3.3.5.2). `LDS` est utilisé avec succès pour résoudre le problème du `VRP` [BH06].

AdjustSize La fonction `AdjustSize` permet au LNS d'adapter le nombre de décisions à relaxer par `RelaxSolution`. Généralement, cela consiste à augmenter le nombre de décisions relaxées lorsque la recherche commence à stagner. Le nombre de décisions relaxées peut aussi être réduit lorsque la recherche progresse bien, ce qui indique que le coût des solutions peut être amélioré avec moins d'efforts de calcul.

5.1.2 LNS et contraintes

La constatation initial de Paul Shaw est que la programmation par contraintes est particulièrement adaptée pour résoudre les problèmes faisant appel à de nombreuses contraintes additionnelles. Ce genre de problèmes est courant dans le monde réel. Cependant les problèmes d'échelle restent une limitation forte. La proposition de Paul Shaw est donc d'utiliser la programmation par contraintes dans une recherche locale. Le but est de profiter de l'efficacité des contraintes sur des sous-parties du problème. La difficulté est

de faire cohabiter une recherche locale, dont l'exploration de l'espace de recherche fonctionne par visite du voisinage, et une recherche par contraintes dont l'exploration repose sur une recherche arborescente. Le LNS consiste à relaxer une partie de l'instanciation et de la réoptimiser. Du point de vue de la recherche locale la *relaxation - réoptimisation* peut être vue comme une exploration et un déplacement dans le voisinage. L'avantage est que l'exploration du voisinage peut être réalisé en utilisant la programmation par contraintes.

Lors des différentes évaluations réalisées par Paul Shaw, LNS présente deux caractéristiques intéressantes. Premièrement, cette méthode est efficace lorsque le problème présente de nombreuses contraintes additionnelles. Deuxièmement, LNS par son principe d'optimisation locale permet de trouver des solutions relativement proches de la solution initiale ce qui permet d'avoir des solutions relativement stables.

5.1.2.1 Choix du sous-ensemble

Le choix d'un sous-ensemble de variables à relaxer est difficile puisqu'il répond à deux besoins contradictoires. Le sous-ensemble doit être suffisamment grand pour permettre à la phase de réoptimisation de s'éloigner suffisamment de la solution initiale pour ne pas rester piégé dans un minimum local. Mais le sous-ensemble doit être suffisamment petit pour permettre à la phase de réoptimisation d'être rapide. Donc un point essentiel pour le choix du sous-ensemble est de choisir des variables qui lorsqu'elles sont relaxées en même temps autorisent de nombreuses nouvelles solutions potentielles. Ainsi une large part des articles [Sha98, PSF04] est consacrée aux méthodes utilisables pour évaluer l'interdépendance des variables.

Approche ad-hoc La première méthode proposée par [Sha98] est assez naïve et se base uniquement sur des raisonnements spécifiques au problème qu'il traite. Ainsi pour déterminer des sous-ensemble de variables à relaxer dans le cadre du `VRP`, il décide de choisir les visites en fonction de leur distance géographique et du véhicule qui doit passer par ce site.

- Plus des sites sont proches plus ils sont supposés liés puisqu'en remettant en cause en même temps l'ensemble des choix fait sur un secteur géographique donné on permet de tester des solutions alternatives avec des sites jouant des rôles similaires.
- De la même manière le second critère repose sur le principe que si les sites sont desservis par le même véhicule c'est parce qu'ils jouent des rôles similaires et que le seul moyen pour réduire le nombre de routes et de remettre en cause la totalité des visites assurées par un véhicule.

Afin de mesurer la force du lien entre deux variables i et j, ils définissent la fonction $\mathcal{R}(i,j) = \frac{1}{c_{ij}+\delta_{ij}}$ avec c_{ij} le coût pour rejoindre j depuis i (la distance dans le cadre du VRP) et δ_{ij} prenant la valeur 1 si i et j sont visités par le même véhicule, 0 sinon. Ainsi lors du choix du sous-ensemble des visites à

Algorithm 4 RelaxSolution(s, $size$)

```
1: decisions := SetOfAllDecisions(s) ;
2: relaxedDecisions := ∅ ;
3: if size > 0 then
4:    d := ChooseRandom(decisions) ;
5:    decisions := decisions \ d ;
6:    relaxedDecisions := relaxedDecisions ∪ d ;
7:    Relax(d) ;
8: end if
9: while —relaxedDecisions— ¡ size do
10:   d := ChooseRelatedDec(decisions, relaxedDecisions) ;
11:   if d ∉ decisions then
12:      d := ChooseRandom(decisions) ;
13:   end if
14:   decisions := decisions \ d ;
15:   relaxedDecisions := relaxedDecisions ∪ d ;
16:   Relax(d) ;
17: end while
```

relaxer, le LNS choisira un site au hasard et relaxera un sous-ensemble des variables qui lui sont le plus fortement liées. La construction du sous-ensemble peut être paramétrée via deux valeurs : le paramètre *size* qui correspond à la taille du sous-ensemble et le paramètre *randomness* qui indique si le choix des variables à relaxer doit suivre strictement la mesure $\mathcal{R}(i, j)$ ou si certaines peuvent être choisies aléatoirement.

Le paramètre *size* est géré de manière incrémentale. En effet le but est d'avoir un ensemble le plus petit possible, mais suffisamment grand pour permettre de couvrir une large part de l'espace de recherche (seul moyen de s'échapper d'un minimum local). Initialement, *tsize* est fixé à 1, durant la recherche, si α itérations ne mènent à aucune amélioration on peut supposer que la recherche est coincée dans un minimum local, on incrémente alors *size* et on relance la recherche. On augmente ainsi la valeur *size* jusqu'à atteindre une valeur maximale qui sert de condition d'arrêt.

Mesure générique statique Dans [HG95], les auteurs définissent une mesure générique pour évaluer le lien entre deux variables. Pour cela, ils introduisent deux fonctions $con(r, u)$ et $tog(r, u)$ portant sur une variable relaxée et sur une variable fixée. La fonction $con(r, u)$ mesure le lien existant entre deux décisions r et u. Si une contrainte porte sur les deux décisions alors $con(r, u)$ renvoie une valeur différente de zéro. Plus la contrainte, dans laquelle apparaissent u et r est restrictive, plus la valeur renvoyée est élevée. La fonction $tog(r, u)$ indique les décisions qui doivent être relaxées ensemble pour permettre de réduire le coût. Ainsi, dans le cadre du VRP, $con(r, u) = 4s + 0.5p$

où s et p sont des variables 0-1 indiquant si les visites r et u appartiennent à une contrainte les forçant à être sur la même route ou à une contrainte de précédence. La fonction $tog(r, u)$ est basée sur la distance séparant les visites, sur la base que deux visites proches peuvent plus facilement s'intervertir pour améliorer la qualité de la solution. D'où $tog(r, u) = \frac{1-d_{r,u}}{D}$ avec D une estimation de la distance maximum entre deux visites. Cette mesure ne se base essentiellement que sur des hypothèses portant sur le type de problème qu'on essaie de traiter. Elle prend peu en compte la structure réelle de l'instance.

Mesure générique dynamique (par réduction) Dans [PSF04], une stratégie, Propagation Guided Large Neighborhood Search (PGLNS), reposant sur la relaxation de variables liées structurellement dans le problème est présentée (Algorithme 5). La proximité entre plusieurs variables est détérminée en utilisant l'impact de Refalo [Ref04]. Cette stratégie permet de définir de manière générique le voisinage d'une solution qui est un élément critique pour l'efficacité d'une recherche locale.

Algorithm 5 Propagation Guided LNS

```
1: while Fragment size greater than desired size do
2:    if Variable list empty then
3:       Choose unbound variable randomly
4:    else
5:       Choose variable in variable list
6:    end if
7:    Freeze variable and propagate
8:    Update variable list
9: end while
```

Le principe de la stratégie `PGLNS` est d'utiliser la propagation pour définir un voisinage. Initialement, toutes les variables sont relaxées, on choisit alors une variable v_0 au hasard et on lui affecte la valeur qu'elle avait dans la solution précédente. La programmation par contraintes permet alors de filtrer les valeurs dans le domaine des autres variables. La stratégie `PGLNS` suppose que les variables fortement liées à la variable v_0 sont les variables ayant perdu beaucoup de valeurs dans leur domaine suite à l'affectation de v_0. A l'étape suivante, une variable détectée comme liées à v_0 est fixée. L'opération est réitérée jusqu'à ce que le nombre de variables restant non-fixées soit suffisamment petit pour permettre une recherche.

Mesure générique dynamique (par expansion) Une autre stratégie (Algorithme 6) proposée pour la construction des sous-ensembles de variables relaxées est celle du *Reversing PGLNS* [PSF04]. La stratégie `PGLNS` construit les voisinages par réduction et en raisonnant sur les variables qu'il souhaite fixer. Autrement dit, rien ne garantit que les variables restantes seront liées

entre elle et que le sous-ensemble sera intéressant à explorer. Pour combler cette lacune, le `Reversing PGLNS` raisonne par expansion du sous-ensemble des variables relaxées et non pas par réduction. Il utilise la mesure de *proximité* basée sur le nombre moyen de valeurs retirées du domaine d'une variable (lorsqu'une autre a été fixée) défini pour la stratégie `PGLNS`. Ainsi, chaque phase de `PGLNS` précède l'utilisation du `Reversing PGLNS` afin de connaitre les liens d'interdépendance entre les variables.

Cette stratégie a été améliorée en ajoutant une correction ϵ à la taille *size* du sous-ensemble de variables considérées. Ainsi, à chaque itération on recherche un sous-ensemble de taille $\epsilon \times size$. La correction ϵ est mise a jour en observant la taille du sous-ensemble trouvé en fonction de la taille du sous-ensemble recherché. Si la taille du sous-ensemble recherché est trop petite par rapport aux sous-ensembles trouvés, cela signifie que le nombre de décisions relaxées n'est pas suffisant pour prendre en compte toutes les variables fortement interdépendantes. La valeur ϵ est donc augmenté. Inversement lorsque les sous-ensembles recherchés sont trop grands par rapport aux sous-ensembles trouvés, cela signifie qu'on cherche à relaxer en même temps des variables n'ayant pas de liens. La valeur de ϵ est donc diminuée.

Les expérimentations réalisées par [PSF04] montrent que le `PGLNS` est globalement stable mais qu'il n'est efficace que sur les problèmes de petite taille. Pour des problèmes trop grand l'information issue de la propagation n'est pas suffisante pour être compétitif avec des structures de voisinage ad-hoc. En revanche le `Reversing PGLNS` avec correction est compétitif avec les voisinages ad-hoc même pour les très grands problèmes. Cela est dû au fait que la valeur ϵ permet d'adapter au mieux la taille du sous-ensemble : dans un premier temps la valeur ϵ augmente et permet d'explorer l'espace de recherche pour trouver une solution potentielle intéressante, puis ϵ diminue afin de se concentrer intensivement sur certaines sous-parties.

Algorithm 6 Reverse Propagation Guided LNS)

```
1: while Fragment size smaller than desired size do
2:   if Variable list empty then
3:     Choose variable randomly
4:   else
5:     Choose variable in variable list
6:   end if
7:   Add variable size to the fragment size
8:   Update variable list with variables close to the selected variable
9: end while
```

La mesure de proximité utilisée par `PGLNS` et `Reversing PGLNS` prend en compte l'influence directe de la dernière décision dans le retrait d'une valeur. Or dans le cas des réseaux de contraintes n-aires, le retrait d'une valeur fait souvent intervenir un ensemble de décisions (explications). Nous nous

sommes donc intéressés aux explications pour développer une mesure d'interdépendance entre les variables.

5.1.2.2 Explication et voisinage

[Ref04] caractérise l'impact d'une décision ($x_i = a$) au travers de la réduction moyenne de l'espace de recherche engendrée par cette décision. Cependant, cette réduction n'est pas dû uniquement à la dernière décision prise. Une réduction peut être le résultat de la conjonction de plusieurs décisions, qui combinées entrainent le retrait d'une valeur. On appelle ces conjonctions des explications de retrait. L'utilisation d'explications peut fournir une information supplémentaire sur l'implication réelle d'une décision dans un retrait. Une décision passée $x_i = a$ a un impact effectif (du point de vue du solveur) sur une valeur val d'une variable x_j si elle apparait dans l'explication justifiant ce retrait. Ainsi [CJ06] étudie plusieurs mesures possibles pour quantifier l'impact d'une décision en se basant sur les explications (nombre de fois qu'une décision apparait dans les explications calculées pour la valeur val de x_j en prenant également en compte la taille de l'explication). Leur étude permet de conclure à l'intérêt que représente les explications pour identifier les relations existant entre les variables d'un problème. Nous avons donc décidé d'exploiter les explications et ces mesures dans le cadre de `LNS`.

5.1.2.3 Explanation Guided LNS

Notre but est de reprendre le principe `PGLNS` en l'adaptant à l'utilisation des explications. Le `PGLNS` mesure le lien entre deux variables, x_i et x_j, en observant le nombre de valeurs retirées dans le domaine de x_j lors de l'instantiation de la variable x_i. Cette mesure ne permet pas d'identifier l'influence des différentes décisions dans le retrait d'une valeur. Seule la dernière décision est prise en compte. L'utilisation des explications permet de connaitre l'ensemble des décisions intervenant lors du retrait d'une valeur. On définit la mesure :

$$I(x_i, x_j) = \sum_{val \in dom(x_j)} \left(\sum_{\epsilon \in E_j^{val}, x_i = a \in \epsilon} \frac{1}{|\epsilon|} \right)$$

Le lien entre deux variables, x_i et x_j, est proportionnel aux nombres de retraits de valeurs du domaine de x_j dans lesquels x_i est impliquée. En effet, le fait que de nombreuses valeurs soient retirées de x_j, à cause de x_i, indique que x_i a une grande influence sur x_j. En outre, plus le nombre de décisions impliquées dans le retrait d'une valeur est faible, plus l'influence des décisions ayant participés est élevée.

Notre approche pour construire un voisinage consiste donc à choisir une variable x de manière aléatoire et de l'instancier ($x = a$). Suite à l'instanciation, la propagation survient. On utilise ensuite la mesure $I(x, y)$ pour déterminer

la variable suivante à fixer. On répète l'algorithme jusqu'à ce que le nombre de variables y restant soit suffisamment petit pour lancer la recherche.

5.1.2.4 Reverse Explanation Guided LNS

Le `EGLNS` basé sur l'explication souffre du même défaut que le `PGLNS` initial. En effet le `EGLNS` définit le voisinage par réduction. Ainsi, on fixe des variables ayant un lien entre elles mais on ne connaît pas le lien existant entre les variables restantes. Le but du *reverse* est de construire le voisinage par expansion. L'algorithme consiste donc à choisir une première variable aléatoirement et à l'ajouter à l'ensemble des variables relaxées. On choisit les variables suivantes en fonction de leur proximité, au sens de $I(x, y)$, avec les variables appartenant à l'ensemble des variables relaxées. L'ensemble des variables relaxées définit le voisinage. Une fois que l'ensemble est suffisamment grand on lance la méthode de reconstruction.

5.1.3 Strategies de réinsertion

La réoptimisation consiste à rechercher grâce aux techniques de programmation par contraintes (heuristiques de choix de variables et valeurs, propagation, filtrage, ...) de nouvelles valeurs pour les variables relaxées qui vont permettre de diminuer le coût de la solution (branch and bound). Afin de favoriser les solutions proches de la solution initiale, [Sha98] propose d'utiliser une recherche de type LDS [HG95]. En paramétrant la *divergence* autorisée par rapport à la solution, on peut favoriser les zones de l'espace de recherche dans lesquelles les solutions ont un grand nombre de couples (variable, valeur) identiques.

Le LNS a le double avantage d'être efficace et d'assurer une relative stabilité des solutions. Il peut donc être intéressant de l'adapter au cadre dynamique. L'adapter à ce cadre implique de pouvoir gérer les relaxations et les restrictions.

5.1.3.1 Le LNS dans un cadre dynamique

L'intérêt de LNS dans le cadre dynamique tient au fait qu'il raisonne à partir d'une solution qu'il essaie de réparer et d'améliorer. Il s'agit précisément du problème auquel on est confronté lorsqu'on traite un problème dynamique. En effet, lorsque le problème évolue on chercher à retrouver une bonne solution au CSP en réparant les solutions précédentes.

Ainsi, prenons le cas d'un CSP_i pour lequel on connaît une solution s_i et auquel on ajoute une variable v. On appelle $CSP_{(i+1)}$ le CSP obtenu en ajoutant la variable v au CSP_i. Notre but d'utiliser la solution s_i pour construire un solution $s_{(i+1)}$ du $CSP_{(i+1)}$. Deux approches sont possibles. Si on peut facilement construire une solution s_{i+1} du $CSP_{(i+1)}$ à partir de s_i. Il suffit d'utiliser `LNS` à partir de s_{i+1} et relaxant en priorité les décisions possédant

un lien avec v afin d'explorer le voisinage à la recherche d'une solution peu coûteuse. S'il est difficile de trouver une solution réalisable s_{i+1}, une autre approche consiste à utiliser `LNS` avec une fonction de coût basé sur le nombre de décisions inconsistantes. On va donc commencer par instancier l'ensemble des variables aux valeurs de s_i, seule la variable v va rester libre. Suivant le principe du `LNS` on va ensuite déterminer un ensemble de variables à relaxer en identifiant celles qui posent le plus de problème avec la variable v. Une fois une solution consistante trouvée, on l'utilise pour lancer `LNS` avec la fonction de coût qu'on cherche à minimiser.

De part son principe de *relaxation-réaffectation*, `LNS` permet, non seulement, de trouver un ensemble de solution stables en exploitant les liens entre les variables, mais surtout d'adapter une solution pour prendre en compte les évolutions du modèle. Il peut donc s'agir d'une réponse intéressante au cas des problèmes dynamiques.

5.2 Evaluation

Afin d'évaluer la pertinence des voisinages calculés à l'aide de la mesure de Refalo et de ceux calculés à l'aide des explications, nous nous sommes intéressés au problème du multiknapsack. On dispose ainsi d'un ensemble d'éléments qu'on souhaite ranger dans un sac à dos. Chaque élément possède une utilité et des coûts. Le but est de maximiser l'utilité totale des objets emportés dans le sac à dos, sans dépasser les différentes limites de coûts. Nous avons considéré une instance portant sur 28 éléments et possédant 10 coûts différents (mknap1-4) et une instance portant sur 50 éléments et possédant 5 contraintes de coûts (mknap1-6).

Dans notre expérimentation nous avons considéré deux sortes de voisinages ceux construis en relaxant un sous ensemble de variables reliées d'après l'impact de Refalo et ceux construis en relaxant un sous ensemble de variables apparaissant fréquemment ensemble dans les explications d'échec. Chaque résultat est la moyenne obtenue sur 20 exécutions de la recherche locale. Nous avons fait varier le nombre d'itérations (c'est à dire le nombre de phases de relaxation-reoptimisation) et la taille des voisinages considérés (nombre de variables relaxées).

On observe (tableau 5.1) que le critère déterminant est la taille des voisinages mais on peut, malgré tout, trouver des solutions proches de l'optimum si on réalise suffisamment d'itérations. On note aussi que le voisinage basé sur les explications est en moyenne plus efficace que celui basé sur Refalo. Dans le cas du problème de multiknapsack cela s'explique en partie par le fait que l'impact de Refalo ne permet pas de détecter les interactions entre les variables alors que dès les premières itérations les explications permettent déjà d'identifier des sous groupes de variables fortement liées.

Instance	Iterations	Taille	Refalo	Explication	Maximum
mknap1-4	5	10	4198	9612	12400
mknap1-4	10	10	6980	9543	12400
mknap1-4	5	20	8868	11605	12400
mknap1-4	10	20	9408	11846	12400
mknap1-6	5	15	8196	12360	16537
mknap1-6	10	15	9120	14112	16537
mknap1-6	5	20	11205	13487	16537
mknap1-6	10	20	11564	14229	16537

FIGURE 5.1: Comparaison des valeurs maximums trouvées en fonction de la taille des voisinages et du nombre d'itérations pour une recherche locale utilisant un calcul du voisinage basé sur l'impact de Refalo ou basé sur les explications.

5.3 Conclusion

Le LNS semble être un bon candidat pour traiter les problèmes dynamiques. En effet, elle permet de combiner la programmation par contraintes et une recherche locale, c'est à dire une approche permettant de gérer de nombreuses contraintes additionnelles et de définir de manière dynamique un voisinage à une approche permettant d'intégrer de manière simple les évolutions du modèle et de favoriser la stabilité. Des expérimentations plus approfondies sont nécessaires pour évaluer l'efficacité de cette approche et en particulier l'utilisation des explications pour la définition du voisinage.

Chapitre 6

Nogoods et Explications

Les problèmes de satisfaction de contraintes sont utilisés pour modéliser et résoudre une large gamme de problèmes. Pour être au plus près du réel, de nombreux problèmes impliquent de raisonner dans des environnements dynamiques. C'est à dire dans des environnements dans lesquels la connaissance n'est pas fixée une fois pour toutes, elle peut évoluer de manière incrémentale entre des requêtes successives.

Nous nous intéressons donc problème du maintien des solutions dans le cadre de la programmation par contraintes. En pratique ce problème peut apparaitre lorsque la connaissance représenté par le CSP peut être modifié par l'utilisateur (résolution interactive des problèmes), par un autre processus (résolution de problèmes distribués) ou par une perturbation extérieur (événement inattendu). Dans ces trois cas il s'agit de maintenir l'ensemble des solutions lorsqu'une contrainte est ajoutée ou supprimée du CSP. Afin d'éviter le travail redondant durant la recherche, il est nécessaire de stocker des informations supplémentaires sur la propagation et les causes du filtrage. Cette information est stockée sous la forme de *nogoods* aussi appelés explications. Nous nous intéressons donc à l'utilisation des nogoods dans le cadre des problèmes dynamiques.

6.1 Utilisation des *nogoods* dans un problème dynamique

La majorité des algorithmes de satisfaction sont écrits pour des CSP statiques. Aussi l'ajout et le retrait de contraintes dans un CSP et la recherche de nouvelles solutions après une modification avec l'un de ces algorithmes entraine un surcoût et la répétition de travail déjà fait. Le but est d'exploiter les propriétés des CSP pour éviter, autant que possible, la répétition.

Le cadre CSP est monotone (les restrictions suppriment des solutions et les relaxations créent de nouvelles solutions). Soit $(X, C_n - c)$ une relaxation du CSP (X, C_n) et $(X, C_n + c)$ une restriction du CSP (X, C_n) ainsi :

1. Si un affectation $\mathcal{A}$ est une solution de $\mathcal{P}_n = (X, C_n)$ alors il est aussi solution de $(X, C_n - c)$
2. Réciproquement, si un affectation $\mathcal{A}$ n'est pas une solution de $\mathcal{P}_n = (X, C_n)$ alors il n'est pas solution de $(X, C_n + c)$. Autrement dit, une contrainte c' induite par $\mathcal{P}_n$ est aussi induite par $(X, C_n + c)$.
3. Finalement, si le CSP (X, C_n) est inconsistant, alors la contrainte $\neg c$ [1] est induite par le CSP $(X, C_n - c)$ et peut être ajouté à $(X, C_n - c)$ avant la recherche.

1. La contrainte tel que $X_{\neg c} = X_c$ et $R^{\neg c}$ est le complémentaire de R_c dans $\Pi_{x \in X_c} Dom(x)$

Le but est d'améliorer l'exploration et d'empêcher le travail redondant grâce à un mécanisme qui va construire une description de la frontière de l'espace exploré. Ces descriptions sont appelées *nogoods*.

Définition 9. Un *nogood* est une paire $(\mathcal{A}, J)$ où $\mathcal{A}$ est un affectation de valeurs à un sous-ensemble des variables d'un CSP et $J \in C$ tel qu'aucune solution du CSP (X,J) ne contienne $\mathcal{A}$. Autrement dit la contrainte interdisant $\mathcal{A}$ est induite par (X, J). J est appelé la justification du nogood.

Connaître chaque *nogood* d'un CSP (X, C) peut être extrêmement intéressant, dans le cadre statique et dynamique, puisque il permet de découvrir l'ensemble des affectations ne pouvant participer à une solution d'un CSP (X, C') avec $C' \subset C$, et certains des affectations impossibles pour un CSP (X, C') avec $C \subset C'$. Le nombre de *nogoods* peut croitre de manière exponentielle avec le nombre de contraintes et de variables. En utilisant la relation d'inclusion on peut ordonner les *nogoods* afin de réduire la complexité spatiale.

Définition 10. Soit $(\mathcal{A}, J)$ un nogood, et $(\mathcal{A}', J')$ tel que $J \subset J'$ et $\mathcal{A} \subset \mathcal{A}'$. Alors $(\mathcal{A}', J')$ est un nogood. On le note $(\mathcal{A}, J) \prec (\mathcal{A}', J')$.

Définition 11. Soit $(\mathcal{A}, J)$ un nogood, $X_J = \cup_{j \in J} Xj$. $\mathcal{A} \downarrow X_J$ la projection de $\mathcal{A}$ sur X_J. Alors $(\mathcal{A} \downarrow X_J, J)$ est un nogood.

Chaque *nogood* construit $(\mathcal{A} \downarrow X_J, J)$ peut être utilisé dans trois buts :
- il est possible d'utiliser les *nogoods* pour le backtrack intelligent [Pro93, GM94].
- la contrainte $\mathcal{A} \downarrow X_J$ induite par (X, J), $J \subset C$, est aussi induite par (X, C). Il donc possible de l'ajouter au CSP afin d'interdire des branches pas encore explorées.
- durant la résolution d'un CSP (X, C'), et pour chaque *nogood* $(\mathcal{A}, J)$ construit tel que $J \subset C'$, la contrainte interdisant $\mathcal{A}$ peut être ajoutée au CSP avant le début de la recherche.

En fonction du but recherché on favorisera une certaine forme des nogoods. En effet, dans le cadre du backtrack intelligent on s'intéresse uniquement aux *nogoods* portant sur les variables impliquées dans le chemin de décision courant, alors que lorsque les *nogoods* sont utilisés pour le filtrage on privilégie des *nogoods* portant sur des variables ne faisant pas partie du chemin de décision.

Pour les CSP dynamiques on cherche à interdire le travail redondant lors d'ajouts/retraits en décrivant les frontières de l'espace de recherche déjà exploré. Ainsi si suite à un événement extérieur nous devons restreindre un CSP (X, C) en ajoutant une contrainte c et pour lequel nous connaissons un ensemble de *nogoods* $\mathcal{N}$, nous poserons le CSP $(X, C + c + c(\mathcal{N}))$ où $c(\mathcal{N})$ est la contrainte interdisant les *nogoods* $\mathcal{N}$. La relaxation est plus compliquée puisqu'elle peut remettre en cause l'espace de recherche déjà exploré et rendre des *nogoods* consistants. Pour détecter les *nogoods* devenus consistants ils doivent

être accompagnés de leur justification (des contraintes qui sont responsables de leur inconsistance). Ainsi si suite à une événement extérieur nous devons relaxer un CSP (X, C) en retirant une contrainte c et pour lequel nous connaissons un ensemble de nogood $\mathcal{N}$, nous poserons le CSP $(X, C - c + c(\mathcal{N}'))$ où $\mathcal{N}^{\neg c}$ est l'ensemble des *nogoods* dans lesquels n'intervient pas la contrainte c.

Dans la suite du chapitre nous nous sommes concentrés sur la gestion de la contrainte $c(\mathcal{N})$ (dans le cadre de la restriction) afin de la rendre efficace et dynamique. En effet les approches classiques imposent de poser un nouveau CSP après chaque modification. Nous nous sommes intéressés à l'utilisation des automates pour pouvoir gérer, durant la recherche l'ensemble des *nogoods* (ajout / retrait).

6.2 Autour des nogoods

L'utilisation des *nogoods* est une technique bien connue pour réduire le thrashing (calcul redondant) rencontré par les algorithmes de recherche arborescent. Un *nogood* peut être défini comme un affectation partiel des variables qui ne peut pas être étendu à une solution [Dec90]. Les *nogoods* ont été initialement introduits en Programmation par Contraintes (PPC) [Dec90, SV94] sans mener à des améliorations significatives des performances à cause d'une consommation de la mémoire qui peut être exponentielle. Cependant, ils sont rapidement devenus un élément efficace des méthodes de résolution SAT. Les solvers SAT semblent traiter correctement les besoins en mémoire et en espace de l'enregistrement des *nogoods* et des travaux plus récents [KB03, KB05] ont déjà tenté de les utiliser dans un contexte PPC. Nous avons cherché comment l'enregistrement de *nogoods* pourrait être appliqué efficacement en PPC.

Chaque échec d'un algorithme de backtrack peut être associé à un *nogood* en considérant les décisions qui ont mené a cet échec. Le principe de l'enregistrement de *nogood* repose sur le fait qu'une contradiction rencontrée pour un certain ensemble de décisions est aussi valable pour un autre ensemble de décisions, proches de celui qui a causé la contradiction. Aujourd'hui, les solveurs de contraintes peuvent découvrir la même inconsistance encore et encore, cette recherche inutile est appelé *thrashing*. En analysant attentivement les causes de l'échec, on peut identifier les décisions responsables de l'échec et les éviter lors des explorations futures.

La première question au sujet de l'enregistrement de *nogoods* est donc : *comment calculer les* nogoods *?*. Pour être efficace il faut être capable de calculer des *nogoods* correspondant à des larges parties de l'espace de recherche qui a été visité. Les *nogoods* ont été utilisé pour le backtrack intelligent ou les CSP dynamiques [SV94], ils font toujours référence aux décisions courantes et peuvent ne pas être très pertinent pour filtrer.

La deuxième question au sujet des nogoods, que nous considérons, est *comment traiter les* nogoods *?* Les *nogoods* ont souvent été utilisés en PPC pour

vérifier si le nœud courant pouvait être étendu à une solution ou non. Les solveurs SAT vont un peu plus loin en appliquant un simple (mais efficace) algorithme de propagation (propagation unitaire) à partir des *nogoods* stockés. De plus, en stockant un *nogood* après chaque échec durant la recherche nous avons à régler le problème du stockage d'un nombre exponentiel de nogoods. La communauté SAT a développé des structures de données intelligentes et efficaces pour stocker et réaliser la propagation unitaire sur un nombre important de clauses ou de nogoods. La technique des *two literals watching* [MMZ+01] est une des techniques les plus concluant pour réaliser la propagation unitaire de maniéré efficace et a été appliqué à la PPC [KB03]. Cependant la mise en place de structure de données dans le contexte PPC est toujours une question ouverte. Le niveau de consistance maintenu par SAT est généralement plus faible que de l'arc consistance ainsi le *thrashing* peut être beaucoup plus couteux puisque les échecs sont découverts plus tardivement. Donc il n'est pas évident que les *nogoods* puissent être avantageux dans le cadre PPC.

Nous nous sommes concentrés sur le problème de gestion des nogoods, les stocker et les propager. Nous avons essayé de coder les *nogoods* sous une forme compacte et canonique dans un automate et de réaliser la consistance d'arc.

Premièrement, nous rappelons comment calculer les *nogoods* et comment les utiliser. Nous présentons ensuite notre solution pour le stockage et pour réaliser la propagation sur un ensemble important de tuples. Puis nous donnons nos premiers résultats expérimentaux.

6.3 *Nogoods* et programmation par contraintes

Un problème de satisfaction de contraintes (CSP) est défini par un tuple $< X, D, C >$ où $X = \{x_1, \ldots, x_n\}$ est un ensemble de variables, $D = \{D_1, \ldots D_n\}$ est l'ensemble des domaines correspondants et d, la taille maximum des domaines. On note D_i^{orig} le domaine initial de x_i tandis que D_i est le domaine courant à n'importe quel moment de la résolution. Et C correspond l'ensemble des contraintes initiales du problème. La résolution d'un CSP est réalisée en des phases d'énumérations et des phases de propagation. L'énumération peut être vue comme l'ajout dynamique de contrainte au problème (contraintes de décision). Pour des raisons de simplicité, nous avons limité l'énumération au cas des contraintes d'assignation de la forme $x_i = a$.

6.3.1 Comment calculer des *nogoods* ?

Explications. Un *nogood* est calculé en analysant les causes d'un échec et en dérivant le coupable d'un sous-ensemble des assignations faites jusqu'à là. Un *nogood* est lié au concept d'explications qui ont été utilisé en programmation par contraintes pour améliorer les algorithmes de backtrack. On introduit les définitions des bases :

Définition 12. Une déduction $(x \neq a)$ est le retrait de la valeur a du domaine de la variable x

Définition 13. Une explication généralisée, $g\_expl(x_i \neq a)$ pour la déduction $(x_i \neq a)$ est définie par deux ensemble de contraintes :
- $C' \subseteq C$
- Δ un ensemble de déductions

tel que $C' \wedge \Delta \wedge (x_i = a)$ est globalement inconsistant.

L'ensemble Δ d'une explication généralisée $g\_expl(x_i \neq a)$ correspond à $g\_expl_\Delta(x_i \neq a)$. Chaque déduction est dûe à d'autres déductions et aux inférences faites durant la propagation. On peut remonter cette chaine de déductions / inférences jusqu'à la décision initiale.

Ainsi on peut calculer une explication depuis une explication généralisée. Un Δ vide pour une déduction $(x_i \neq a)$ réfère au fait que les déductions sont soit dûe directement à une décision prise sur x_i tel que $x_i = b$ ou soit réalisé sur le nœud racine. Expliquer une déduction à l'aide uniquement des décisions est le but des explications classiques :

Définition 14. Une explication $expl(x_i \neq a)$ pour la déduction $(x_i \neq a)$ est définie par deux ensembles de contraintes :
- $C' \subseteq C$
- DC un ensemble de contraintes de décisions (affectations).

tel que $C' \wedge DC \wedge (x_i = a)$ est globalement inconsistant.

Les techniques backtrack intelligent stockent généralement une explication pour chaque déduction [JB97]. De telles explications sont calculées au vol par chaque contrainte et sont stockées au niveau des variables. Cela ajoute de la complexité en temps (les algorithmes de filtrages doivent intégrer une algorithme d'explication) et de la complexité en espace ($O(nd)$ au plus une explication est stockée par valeur [2]).

Exemple. Soit x_1 et x_2 deux variables tels que $D_{x_1} = \{0, \ldots, 6\}$ et $D_{x_2} = \{0, 2, 3, 4, 6\}$. Les valeurs 1 et 5 sont retirées du domaine x_2 donc une explication est déjà disponible pour ces déductions. Supposons, $expl_\Delta(x_2 \neq 1) = \{x_4 = 2\}$ et $expl_\Delta(x_2 \neq 5) = \{x_0 = 3, x_8 = 1\}$ et considérons la contrainte $|x - y| = 2$. La valeur 3 est retirée de x en appliquant l'algorithme de filtrage de $|x - y| = 2$. Une explication généralisée est simplement $g\_expl_\Delta(x \neq 3) = \{y \neq 1, y \neq 5\}$. Une explication est, par exemple, $expl_\Delta(x \neq 3) = expl(y \neq 1) \cup expl(y \neq 5) = \{x_4 = 2, x_0 = 3, x_8 = 1\}$.

Les explications sont prévues pour les algorithmes de backtrack intelligent et font, par conséquence, référence au chemin de décision. En expliquant chaque retrait de valeur, on peut expliquer une contradiction (un domaine vide

2. Il s'agit d'une borne supérieure utilisée pour limiter la consommation en espace.

D_i) en réalisant l'union de toutes les explications de retraits pour chacune des valeurs de D_i^{orig}. L'explication obtenue, $expl(D_i = \emptyset) = \bigcup_{j \in D_i^{orig}} expl(x_i \neq j)$, est souvent appelé une explication de contradiction et correspond exactement à la notion habituelle de *nogood* comme *un ensemble d'affectation qui ne peut pas être étendu à une solution.*

L'idée des *nogoods* généralisés [KB05] est d'améliorer le pouvoir de filtrage des *nogoods* en gardant les raisons immédiates (explications généralisées) d'un retrait plutôt que de les projeter sur le chemin de décision courant (lequel est obligatoire pour le backtrack intelligent et qui est critique pour la réparation dynamique comme dans *dbt* [GM94])) et retarde le calcul du *nogood* lorsqu'un un échec survient. Suivant l'idée de [JB97, KB05], on peut définir un *nogood* généralisé :

Définition 15. Un *nogood* généralisé est un ensemble de contraintes $C^{'}$, un ensemble de déductions Δ et un ensemble de contraintes de décisions DC tel que $C^{'} \wedge \Delta \wedge DC$ est inconsistant.

Le sous-ensemble $C^{'}$ ne sera pas utilisé mais il est utile pour expliquer l'optimalite/inconsistance du problème à l'utilisateur. En stockant des explications généralisées, on garde en mémoire la chaine logique des inférences faites durant la résolution, qui est appelé en SAT, le graphe d'implication [ZMMM01, BHZ05]. Depuis une contradiction causée par un domaine vide D_i de la variable x_i, on peut calculer plusieurs *nogoods* généralisés (tandis qu'un seul *nogood* était disponible avec la technique standard).

La technique générale pour calculer un nogood [3] est donnée par l'algorithme 7. Ligne 1 on commence en calculant le *nogood* généralisé exprimant le fait que la variable x_i a été vidée et a levé une contradiction. Alors chaque déduction peut être remplacée par son explication généralisée pour obtenir un nouveau (et peut être plus riche) nogood. Une explication généralisée dont l'ensemble Δ est vide (ligne 6) est dû à une décision faite sur ses variables aussi nous utilisons la décision correspondante [4]. Un *nogood* généralisé est finalement autant composé de déduction que d'assignation. On peut implémenter n'importe quelle technique SAT d'enregistrement en choisissant la condition d'arrêt correctement. Par exemple, nous allons implémenter `Unique Implication Point` [BHZ05] si nous décidons de nous arrêter lorsqu'on trouve une raison singleton qui implique le conflit au niveau courant de décision.

6.3.2 Comment exploiter les *nogoods* ?

La propagation des *nogoods* est généralement limitée à la propagation unitaire réalisée par les solveurs SAT. Il y a en fait une correspondance étroite

3. Cela équivaut à calculer une coupe dans le graphe d'implication introduit en SAT. Le graphe d'implication est connu en programmation par contraintes comme un arbre de preuve [DFJ+03].

4. L'explication peut être vide si la déduction est réalisée au nœud racine.

Algorithm 7 computeGeneralizedNogood(Var x_i)

```
1: GeneralizedContradictionExplanation e ← ⋃_{j∈D^orig} x_i ≠ j ;
2: while stopping criterion not met do
3:    x_k ≠ k ← choose a deduction of e ;
4:    if g_expl_Δ(x_k ≠ k) is not empty
5:       e ← e ∪ g_expl(x_k ≠ k) − {x_k ≠ k} ;
6:    else e ← e ∪ expl(x_k ≠ k) − {x_k ≠ k} ;
7: end while
8: return e ;
```

avec SAT. Appelons un littéral une paire variable/valeur (x_i, j). Un littéral positif correspond a $x_i = j$ tandis qu'un littéral négatif correspond à $x_i \neq j$. Un littéral positif (respec. négatif) est dit satisfait des que x_i est instancié à la valeur j (respec. j retiré de x_i), insatisfait dans le cas contraire et libre sinon. Un *nogood* généralisé (Δ, DC) peut être vu comme une contrainte, une clause portant sur les littéraux correspondants $(\bigvee_{x_k \neq j \in \Delta} x_k = j) \vee (\bigvee_{x_k = j \in DC} x_k \neq j)$ qui doivent être satisfaits dans le reste de la recherche. Un *nogood* est libre aussi longtemps que deux littéraux sont libres, satisfaits dès qu'un littéral est satisfait et violé lorsque tout les littéraux sont violés. De plus le *nogood* est dit unitaire quand seul un littéral est libre tandis que les autres sont insatisfaits. Dans ce cas la propagation unitaire force le littéral libre a être satisfait.

Les mécanismes de la propagation unitaire employés par les solveurs SAT ont été grandement améliorés et optimisés grâce aux compétitions SAT. Un résumé de ces techniques de propagation peut être trouvé dans [BHZ05]. La technique des *two literals watching* [MMZ+01, BHZ05] est actuellement reconnue comme étant le meilleur moyen pour propager les clauses SAT. Nous introduisons rapidement le procédé puisqu'il servira de base pour la propagation des nogoods. Le statut d'un *nogood* (satisfisait, violé, libre) peut être connu en observant seulement deux littéraux. Chaque *nogood* est *observé* par deux pointeurs sur deux littéraux libres. Deux listes de *nogoods* sont par conséquent maintenu pour chaque littéral, la liste positive $pos\_watch(x_i, j)$ est la liste des *nogoods* ou le littéral positif $x_i = j$ est surveillé et la liste négative $neg\_watch(x_i, j)$ correspondant à la liste des *nogoods* où $x_i \neq j$ est observé. $pos\_watch(x_i, j)$ est parcouru lorsqu'un retrait de valeur $x_i \neq j$ est reçu et $neg\_watch(x_i, j)$ est considéré en cas d'assignation. Pour chaque *nogood* de la liste, les littéraux observés (qui ne sont pas violés) ont besoin d'être mis à jour et plusieurs cas sont possibles :

- L'autre littéral observé est déjà satisfait, le pointeur du littéral violé est laissé inchangé ;
- Un autre littéral libre ou satisfait est trouvé et la liste observé est mise à jour en conséquence ;
- Autrement, tout les autres littéraux sont satisfait. Le *nogood* est unit et l'autre littéral libre est propagé. Le littéral violé est laissé inchangé.

L'astuce des *watched litterals* et de laisser les pointeurs sur les littéraux violés qui vont rester valides lors du retour arrière. En faite, la technique assure que des qu'un *nogood* est libre, il est observé par deux littéraux libres. Ajouté dynamiquement un *nogood* a une feuille de l'arbre de recherche est fait en plaçant les pointeurs sur les derniers littéraux intervenant dans les deductions/décisions ainsi, le *nogood* va être observe correctement lors du retour arriéré. On peut remarquer que la technique *watched literals* est bien adapté lorsqu'on utiliser une grande quantité de mémoire puisque aucune restauration des structures de données n'est nécessaire lors du retour arriéré.

Prendre en compte des *nogoods* peut néanmoins mener a une consommation exponentielle de la mémoire et il est obligatoire d'oublier, de temps en temps, une partie des *nogoods* appris. Plusieurs stratégies ont été introduites pour oublier les *nogoods* comme *i-order bounded learning* [SV94] ou *i-order relevance bounded learning* [BM96]. Ils proposent de se souvenir uniquement des *nogoods* de taille maximale i ou de seulement ceux qui sont toujours pertinent par rapport au chemin de décisions courant (qui ne diffère de pas plus de i éléments du chemin de décision). La complexité spatiale des techniques précédentes est de $O(n \times d^i)$. Cependant ce compromis n'est pas toujours rentable [KB03, LMS02] et un grand nombre de *nogood* doit être enregistré pour rendre l'apprentissage efficace.

Des techniques de propagation efficace ainsi qu'une gestion optimisée de l'espace sont par conséquent des problèmes clefs pour les techniques de stockage des nogoods. Pour prendre en compte la difficulté des nogoods, on propose de les gérer dans une forme compilée comme un automate. Notre hypothèse est que les *nogoods* peuvent partager un grand nombre de littéraux lorsqu'ils sont appris dans le même sous arbre. Enregistrer les *nogoods* n'est pas suffisant et la propagation basée sur les automate doit être réalisée de manière efficace.

6.4 Des automates pour l'encodage des nogoods

Un ensemble de tuples portant sur n variables peut être encodé dans un automate acyclique avec $l = (n + 1)$ couches correspondant à chaque variable et à l'état final F. Mais les *nogoods* généralisés peuvent être considérés comme des tuples. On considère des domaines finis D_i donc une déduction $(x_i \neq v_j)$ peut être vue comme $\vee_{v_k \in D_i \setminus \{v_j\}}(x_i = v_k)$. Un automate fini déterministe est un tuple $(Q, \Sigma, \delta, q_0, F)$ où Q est un ensemble fini d'états et $q_0 \in Q$ est l'état initial. L'alphabet Σ correspond à l'union de tous les domaines des variables et Σ^* l'ensemble de tout les mots. Chaque variable x_i est associée à la i^{eme} couche de l'automate et les transitions sortantes des nœuds appartenant à la couche i sont étiquetées par les valeurs du domaine D_i^{orig}. δ est la fonction de transition de $Q \times \Sigma \mapsto Q$ et $\delta(q, val)$ correspond à l'état atteint en appliquant

la transition val depuis l'état q et la paire (q, val) définit l'arc correspondant [5].

On note $\gamma(q_1, q_2)$ les valeurs de transition qui permettent d'atteindre l'état q_2 depuis q_1.

Un exemple d'un tel automate est donné figure 6.4. Nous noterons $|A|$, le nombre d'états de l'automate A et $|A_i|$ le nombre d'états de la i^{eme} couche.

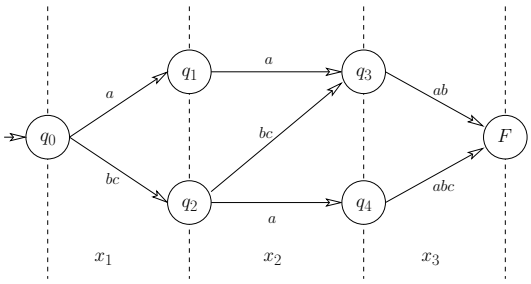

FIGURE 6.1: Un exemple d'un automate portant sur trois variables x_1, x_2, x_3 de domaine $\{a, b, c\}$ encodant les tuples (a, a, a), (a, a, b), (b, b, a), (b, b, b), (b, c, a), (b, c, b), etc.

Cette représentation a déjà été utilisée dans le contexte des problèmes de configuration [AFM02]. L'automate est un moyen générique de représenter un ensemble de tuples et par conséquent de définir une contrainte en extension. Pour un domaine donné, S de tuples sur une séquence finie de variables X, nous nous referons à :

- A l'automate reconnaissant les tuples faisables correspondant à S. C'est à dire, $\mathcal{L}(A) = \{w \in \Sigma^* / \delta^*(q_0, w) = F\} = S$
- $\overline{A}$ l'automate reconnaissant les tuples infaisables correspondant à S. En d'autres mots, $\mathcal{L}(\overline{A}) = \{w \in \Sigma^l / w \notin \mathcal{L}(A)\}$ où Σ^l correspond aux mots de longueur l.
- $A(x)$ l'automate projetant l'état courant du domaine de la variable X *cad* touts les arcs (q, j) pour un état q situé sur la couche i tel que $j \notin D_i$ sont retirés.

Lorsque l'automate est minimisé (pour un ordre donné des variables) il est unique. Un automate est minimal si il n'y a pas d'états équivalents. Deux états sont équivalents si ils devinent le même langage droit : $\overrightarrow{\mathcal{L}}$ (ils ont le même ensemble de mots permettant d'accéder à l'état final depuis eux). Comme nous considérons un automate en couches, nous pouvons le minimiser efficacement en utilisant une méthode bottom-to-top basée sur une définition récursive du

5. δ^* étend δ tel que

$$\delta^*(q, w) = \begin{cases} \delta^*(\delta(q, x), y) & \text{if } w = xy \text{ avec } x \in \Sigma \text{ et } y \in \Sigma^+ \\ \delta(q, w) & \text{si } w \in \Sigma \end{cases}$$

langage droit d'un état :

$$\overrightarrow{\mathcal{L}}(q) = \{a\overrightarrow{\mathcal{L}}(\delta(q,a))/a \in \Sigma \wedge \delta(q,a) \neq \perp\} \cup \begin{cases} \{\varepsilon\} & \text{if } q \in F, \\ \emptyset & \text{sinon,} \end{cases}$$

On peut noter, que dans un automate minimal, coder les tuples faisables ou les tuples infaisables n'a pas d'influence sur la taille de l'automate. On peut facilement prouver que le nombre d'états de A et $\overline{A}$ diffère au plus de l états.

Propriété 1. *Si A et $\overline{A}$ sont minimaux alors $abs(|A| - |\overline{A}|) < l$*

Preuve : On peut montrer comment construire $\overline{A}$ à partir de A (voir figure 6.4). Premièrement, on change l'état final de A en état poubelle afin que tous les tuples valides de A deviennent interdits. Deuxièmement, les tuples invalides de A doivent être reconnus et toutes les transitions manquantes (celles qui allaient implicitement à l'état poubelle) doivent être ajoutées (les arcs en gras sur la figure 6.4). Au plus $(l-1)$ états sont retirés (arcs en pointillés sur la figure 6.4). En fait un état est retiré si toutes ses transitions mènent à l'ancien état final et seul un état par couche peut avoir cette propriété (sinon les deux états seraient équivalents). Encore, au plus un état est ajouté par couche (encore a cause de la minimalité, seul un état peut avoir toutes ses transitions sortantes qui mènent au nouvel état final)

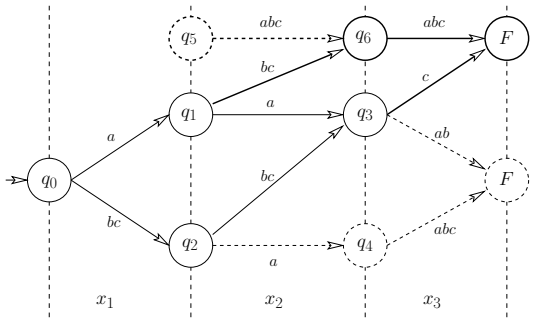

FIGURE 6.2: Transforamtion de A (constitué des arcs normaux et des arcs en pointillés) en $\overline{A}$ (composé des arcs normaux et des arcs en gras)

Considérant un ensemble de *nogood* S (tuples infaisables), on choisit de maintenir l'automate $\overline{A}$ correspondant par conséquent aux tuples faisables. Ajouter un *nogood* dans un automate signifie retirer le mot correspondant du langage reconnu par l'automate. Comme dit précédemment, cela a peu de conséquences sur la taille de l'automate mais il est plus facile de raisonner à partir de $\overline{A}$ lorsqu'on réalise le filtrage.

6.4.1 Minimisation incrementale

On décrit comment la minimisation instrumentale fonctionne pour expliquer les deux stratégies de minimisation. Le but est de maintenir

incrémentalement l'automate des tuples faisables. Nous devons être capable d'ajouter des *nogoods* (retirer les mots correspondant du langage reconnu par l'automate) incrémentalement lorsque nous en découvrons de nouveaux.

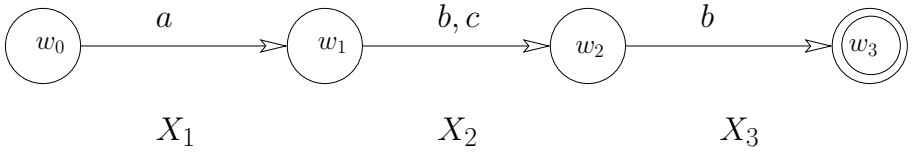

FIGURE 6.3: L'automate chaine qui reconnaît le *nogood* généralisé $(x_1 = a) \wedge (x_2 \neq a) \wedge (x_3 = b)$ with $\Sigma = \{a, b, c\}$

On considéré l'alphabet finit Σ, chaque deduction $(x \neq v)$, faite à partir d'un *nogood*, peut être remplacée par $\{x \in \Sigma \setminus v\}$. Notons w le mot correspondant au *nogood* à retirer des tuples autorisés de $\mathcal{L}(A)$. Retirer w de $\mathcal{L}(A)$ consiste à construire un nouvel automate $A \cap \overline{W}$ avec W l'automate chaine reconnaissant w (voir la figure 6.4.1). W est construit en utilisant le même ordre des variables de A tel que $\delta^*(w_0, w)$ est l'état final. L'algorithme procède en deux étapes (décrites figure 6.4.1) :

- Calculer $A \cap \overline{W}$: les différences principales avec les autres méthodes utilisées pour construire incrémentalement des automates acycliques minimaux est que nous essayons de retirer une chaine au lieu de l'ajouter et un *nogood* généralisé peut représenter plus d'un mot.
- Minimise incrémentalement le nouvel automate en prenant en compte les nouveaux états ajoutés : en utilisant le fait que notre automate est en couches nous pouvons le minimiser de manière efficace.

La complexité temporelle du retrait et de la minimisation est $O(|W| + |\Sigma| \times |W|)$.

Ajouter un *nogood* w peut ajouter au plus w états à l'automate même si aucune minimisation ne survient (voir l'étape b de la figure 6.4.1). Ce n'est pas vrai, cependant, pour les *nogoods* généralisés.

Actuellement un inconvénient, lorsque nous considérons des *nogoods* généralisés, est que l'automate peut être plus grand (en nombre d'états) que la somme du nombre d'états de l'automate chaîne correspondant aux nogoods. C'est dû au fait qu'un *nogood* généralisé représente plusieurs tuples. Un automate chaîne est déjà une forme de représentation compacte. De plus ce comportement est difficile à prédire comme il est difficile la taille de l'automate pour un langage donné. Dans le pire des cas, ajouter un tuple à l'automate peut ajouter $\sum_{k=2}^{l-1} |A_k|$ états. Nous cherchons des heuristiques pour choisir un sous-ensemble des tuples qui peuvent se compacter efficacement dans un automate. De plus, les méthodes basées sur le classement lexicographique des tuples ne peut pas être utilisé dans notre cas à cause des *nogoods* généralisés.

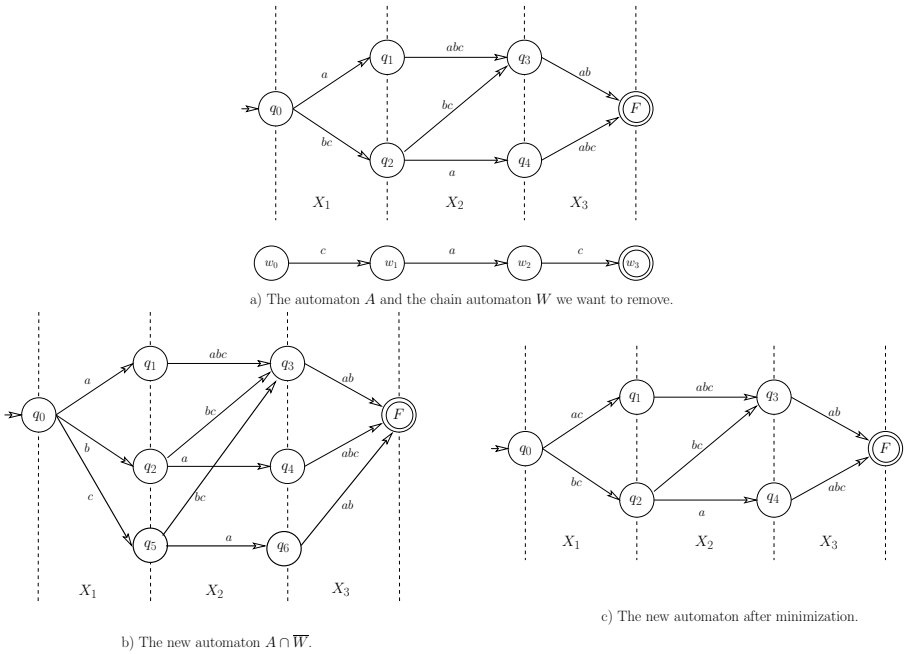

FIGURE 6.4: Processus de minimisation

Un second inconvénient est le problème de trouver un bon ordre pour les variables. La taille de l'automate est liée à l'ordre des variables. Comme les tuples sont découverts durant la recherche, le calcul dynamique de l'automate impliquerait de le réordonner dynamiquement. Aussi pour profiter de l'heuristique d'ordre de variable nous avons décidé de tester une seconde stratégie : au lieu d'ajouter de nouveaux *nogoods* au vol, on calcule l'automate minimal périodiquement dès qu'une limite d'utilisation mémoire a été atteinte. Les heuristiques pour trouver un bon ordre sont basées sur des idées similaires a celles utilisées dans les ROBDD [Bry86] et essaie de regrouper les variables qui sont partagés par un de nombreux nogoods.

6.4.2 Algorithme de filtrage

[Pes04] fournit un algorithme de filtrage pour une contrainte globale décrite par un langage régulier. L'arc-consistance sur A peut par conséquent être réalisé en appliquant l'algorithme de Pesant. L'idée est de maintenir l'ensemble Q_{ij} des états agissant comme des supports pour chaque paire variable-valeur (x_i, v_j). Un état q de la $i$éme couche est considéré comme un support de (x_i, v_j) tant qu'il existe un chemin de q_0 à q et de $\delta(q, v_j)$ à F dans $A(X)$. Une fois Q_{ij} est vide, la valeur j est retirée de la variable x_i.

Sur la figure 6.4, nous avons par exemple $Q_{1a} = Q_{1b} = Q_{1c} = \{q_0\}$,

$Q_{2a} = \{q_1, q_2\}$, $Q_{2b} = Q_{2c} = \{q_2\}$, $Q_{3a} = Q_{3b} = \{q_3, q_4\}$ et $Q_{3c} = \{q_4\}$. La propagation incrémentale est faite en stockant dans une structure de données backtrackable les arcs entrants et sortants pour chaque nœud ainsi que leur dégrés entrant et sortant. Chaque fois qu'une valeur j est retirée de la variable i, les degrés des états dans Q_{ij} sont décrémentés en conséquence. Si un degré atteint 0, l'information est propagée à l'ensemble des nœuds qui lui sont connectés (les prédécesseurs si le degré sortant est nul et successeur si il s'agit du degré entrant) en décrémentant leur degré et en maintenant les listes Q_{ij}.

Sur l'exemple de la figure 6.4, si les valeurs a et b sont retirées de x_3, le degré sortant de q_3 tombe à 0, donc ses arcs entrants sont parcourus. Les états q_1 et q_2 sont retirés de Q_{2a}, Q_{2b} et Q_{2c} durant le parcours des arcs entrants. Comme Q_{2b}, Q_{2c} sont mis à jour à $\emptyset$, les valeurs b et c sont retirées de x_2. De plus, le degré sortant de q_1 et q_2 est décrémenté et le processus continue comme le degré de q_1 atteint 0, ainsi la valeur a est finalement retirée de x_1.

6.4.2.1 Explication de l'algorithme de filtrage basé sur l'automate.

Comme dit dans la section 6.3.1, chaque algorithme de filtrage doit être *expliqué* afin de pouvoir générer des *nogoods* généralisés. Chaque fois qu'une valeur est retirée, une explication généralisée doit être associée à la deduction. Ainsi, il est obligatoire d'expliquer le filtrage provoqué par les *nogoods* compilés dans l'automate. Expliquer le filtrage de l'automate est réalisé en :

- Expliquer pourquoi un état ne peut pas atteindre F (algorithme 8)
- Expliquer pourquoi un état ne peut pas être atteint depuis q_0 (algorithme 9).

Algorithm 8 explain_Why_q_Cannot_Reach_F(State q, int i)

```
1: Explanation e ← ∅ ;
2: if is_explained(q) is false then
3:    for all j such that δ(q, j) ≠ null do
4:       if j ∈ D_i then e ← e ∪ expl(δ(q, j)) ;
5:       else e ← e ∪ expl(x_i ≠ j) ;
6:    end for
7:    is_explained(q) ← true ;
8:    expl(q) ← e ;
9: end if
```

Une explication $expl(q)$ et un booléen backtrackable $is\_explained(q)$, sont attachés a chaque état q de l'automate original. $expl(q)$ enregistre pourquoi q est invalide *cad* pourquoi il ne peut pas être sur un chemin de q_0 à F. $is\_explained(q)$ est vrai si l'invalidité de q a déjà été expliqué et une $expl(q)$ valide est disponible dans la branche courante de l'arbre. Comme plusieurs explications existent, il est obligatoire d'interdire la réécriture d'une explication

Algorithm 9 explain_Why_q_Cannot_ Be_Reached_From_q0(State q, int i)

```
1: Explanation e ← ∅ ;
2: if is_explained(q) is false then
3:    for all (p, j) such that δ(p, j) == q do
4:       if j ∈ D_{i−1} then e ← e ∪ expl(p) ;
5:       else e ← e ∪ expl(x_{i−1} ≠ j) ;
6:    end for
7:    is_explained(q) ← true ;
8:    expl(q) ← e ;
9: end if
```

valide parce que l'explication elle même n'est pas restaurable lors du retour arrière et on pourrait perdre l'explication originelle par la même occasion.

Une valeur j d'une variable x_i est retirée parce que Q_{ij} est vide. On peut expliquer le filtrage parce que pour chaque état q qui faisait partie de la liste originelle des supports de (x_i, v_j) (noté $Qinit_{ij}$), soit q est lui même invalide ou $\delta(q, j)$ est invalide (algorithme 10).

Algorithm 10 prune(int i, int j)

```
1: Explanation e ← ∅ ;
2: for all q in Qinit_{ij} do
3:    if is_explained(q) then e ← e ∪ expl(q) ;
4:    else e ← e ∪ expl(δ(q, j)) ;
5: end for
6: remove value j from x_i due to e ;
```

Le champs $expl(q)$ est calculé pour chaque état q de la façon suivante :

– Pour expliquer pourquoi un état q_k sur la couche ne peut pas être atteint depuis q_0, nous divisons ses prédécesseurs en deux ensembles $rpreds$ et $\overline{rpred}$. Le prédécesseur $rpred$, qui peut être atteint depuis q_0, et ceux, $\overline{rpred}$, qui sont inatteignable. Pour chaque prédécesseur p de q_k, soit il appartient à $\overline{rpred}$ et on utilise l'explication $expl(p)$ attaché à p, ou il appartient à $rpred$ et les valeurs de transitions menant de p à q_k ($\gamma(p, q_k)$) ont été retirées du domaine de x_{i-1}. L'algorithme 8 est appelé chaque fois que le degré entrant de q_k atteint 0 et calcule $expl(q_k)$:

$$expl(q_k) = expl(q_0 \nRightarrow q_k) = \bigcup_{p \in rpred} expl(x_{i-1} \neq \gamma(p, q_k)) \cup \bigcup_{p \in \overline{rpred}} expl(p)$$

– D'une manière similaire, l'état q_k ne peut pas atteindre F parce que soit son successeur ne peut pas atteindre F ou la valeur permettant d'atteindre un état relié à F est manquante. L'algorithme 9 est appelé

chaque fois que le degré sortant de q_k atteint 0 et calcule $expl(q_k)$:

$$expl(q_k) = expl(q_k \nRightarrow F) = \bigcup_{s \in rsucc} expl(x_i \neq \gamma(q_k, s)) \cup \bigcup_{s \in \overline{rsucc}} expl(s)$$

6.4.2.2 Algorithme de filtrage allégé.

Le but de l'automate est de compiler des grands ensembles de *nogoods* et par conséquent être capable de prendre en compte des consommations importantes de mémoire. L'algorithme de propagation incrémentale est dans un sens très consommateur de mémoire puisqu'il requiert deux listes doublement chainées (arcs entrants et arcs sortants) et deux entiers (degré entrant et degré sortant) par état qui sont restaurables lors du retour arrière. Il utilise aussi une liste bactrackable Q_{ij} d'états par paire variable-valeur. Premièrement on abandonne le maintien des listes doublement chainées pour les arcs entrants et sortants. Si le nombre d'arcs sortants est borné par la taille de l'alphabet (la taille du domaine maximale), le nombre d'arcs entrants peut être égal au nombre d'états de la couche précédente, ce qui ne semble pas raisonnable dans notre cas. Cet algorithme est noté $Aut0$ dans la suite. De plus, nous avons étudié le compromis suivant qui réduit le temps constant de mise à jour à chaque retrait de variable-valeur :

- Comme expliqué par Pesant dans [Pes04], nous n'avons pas besoin de tous les supports pour chaque état. Un seul peut être gardé en mémoire.
- Deuxièmement, nous n'avons pas besoin du degré exact de chaque état mais seulement de savoir si le degré est nul ou non.

Une force des watched literals repose précisément dans le fait que rien ne nécessite d'être restauré lors du retour arrière. Nous avons essayé, en utilisant ce principe d'économiser de la mémoire en gardant un arc entrant/sortant par état mis à jour uniquement lorsque l'arc est perdu au lieu de stocker le degré. Un arc valide à la profondeur k dans l'arbre de recherche et aussi valide au profondeur inférieur à k. Le filtrage basé sur $Aut0$ avec les améliorations précédentes est noté $Aut1$. Finalement, nous stockons seulement un état support pour chaque valeur (x_i, v_j). Quand ce support devient invalide, nous cherchons un autre parmi les arcs $Q_{init_{ij}}$. $Aut1$ combiné à cette amélioration est noté $Aut2$.

6.5 Premiers résultats expérimentaux

Nous avons d'abord étudié, l'intérêt de stocker de grandes tables de tuples dans un automate pour réaliser le filtrage en utilisant l'algorithme décrit au dessous en relation avec la méthode classique comme l'algorithme de l'arc consistance généralise décrite dans [BR97]. Nous avons ensuite évalué l'utilité de stocker dynamiquement des *nogoods* en utilisant l'algorithme des watched

literals pour la phase de propagation. On considère les puzzles de mots croisés et des problèmes RLFAP.

6.5.1 Premier aperçu : Stockage et filtrage

Les problèmes des puzzles de mots croisée consistent à remplacer les mots croisés en utilisant des mots fournis dans un dictionnaire de référence et chaque mot doit être utilisé une et unique fois. Ainsi les contraintes doivent stocker de grandes tables de tuples correspondant aux mots autorisés du dictionnaire.

Une variable x_i est associée à chaque case libre du puzzle et son domaine $D(x_i) = \{a, b, \ldots, z\}$ est composé des vingt-six lettres de l'alphabet. Une contrainte est posée par mot, *c-à-d*, par séquences contigues de lettres dans le puzzle. Les tuples autorisés de la contrainte sont définis par tout les mots de la longueur correspondante dans un dictionnaire de référence. Un mot peut être utilisé seulement une fois dans le puzzle donc une contrainte de différence est aussi ajoutée entre chaque paire de mot de la même taille. On étudie deux approches pour réaliser GAC sur le problème :

1. L'algorithme du GAC est introduit par [BR97] [6] pour les contraintes décrites en extension par une table de tuples autorisés. Pour atteindre une efficacité raisonnable, un accès direct au support de chaque paire variable-valeur est donné grâce à une structure de données partagée entre les contraintes. Les listes chaînées de mots ordonnés de taille k ayant une lettre l à une position donnée p sont stockés dans des tableaux à 3 dimensions appelles `supports[l][p][k]`. GAC est réalisé avec l'algorithme GAC2001 en stockant le support courant (un entier backtrackable correspondant à l'index du mot dans la structure de données `supports`).

2. L'algorithme de propagation décrit précédemment. Chaque dictionnaire de taille k (tout les mots de taille k) est compilé dans un automate minimal appelé $auto_k$.

Résultats Nous utilisons les benchmarks de [ABvB01] qui sont composés d'instances de taille 5x5 à 23x23 et proviennent de Herald Tribune Crosswrods. Nous utilisons le dictionnaire *words* qui regroupe 45000 mots (un autre dictionnaire, *UK*, inclus 220000 mots mais les instances sont souvent plus dures avec le dictionnaire *words* même si l'espace de recherche est plus petit parce qu'il contient moins de solutions). La limite de temps est mise à une heure. On ne donne ici que quelques résultats significatifs sur un sous-ensemble des 50 instances avec le dictionnaire *words*.

6. La multidirectionnalité n'est pas implémenté dans notre algorithme du GAC.

Instances	**M**AC-Aut0		**M**AC-Aut1	**M**AC-Aut2	**M**AC-GAC	
(dico :words)	**t**ime (s)	**n**ode	**t**ime (s)	**t**ime (s)	**t**ime (s)	**n**ode
05.01	1	30	1	0,8	**0**,4	30
15.01	1,7	75	1,2	1,3	**0**,9	75
15.02	**1**2,7	872	13,4	19,5	23,9	872
15.07	**3**34,8	22859	374,2	660,1	857,5	22859
19.02	**8**6,8	17511	99,5	240,8	231,5	17511
19.05	> 1h	**1**238063	> 1h	> 1h	> 1h	537235
21.03	**7**1,9	13017	86,3	243,8	223,9	13017
21.06	> 1h	**4**70253	> 1h	> 1h	> 1h	143638
21.07	**2**0,3	1825	23,1	39,0	45,9	1825
23.07	> 1h	**2**43678	> 1h	> 1h	> 1h	110613

1. La propagation initiale de l'automate est très coûteuse (il faut initialiser les supports d'états). GAC2001 est par conséquent plus rapide pour les instances comportant peu de nœuds.
2. Sure les instances dures, l'automate tend à être deux ou trois fois plus rapide que le GAC2001. Ceci est dû au fait que le support d'une lettre l à la position p dans un mot de taille k sont les états de la p-ieme couche de l'automate $auto_k$ qui ont la lettre l dans leurs transitions sortantes. Ce sous-ensemble d'états jouant le rôle de support est noté `state-supports[l][p][k]`. Tous les mots correspondant au support réel peuvent en faite être lus dans l'automate dans un chemin allant de l'état initial à l'état final passant par les états de `state-supports[l][p][k]`. Comme les mots sont des tuples très "structurés" (ils partagent beaucoup d'assignations) nous avons |state-supports[l][p][k]| < |supports[l][p][k]|. Le résultat est que la fonction `seekNextSupport` qui est la base de n'importe quel algorithme de GAC est plus rapide sur state-supports[l][p][k] que sur supports[l][p][k]. Cependant, la multidirectionnalité peut améliorer un peu la méthode GAC2001.

Une autre question intéressante est le problème de la mémoire. Pour stocker tout les mots de taille 8 du dictionnaire *words*, la structure de données `supports[l][p][k]` nécessite 3,98 Méga byte contre 1,70 pour l'automate. L'automate est par conséquent plus compact, bien que que la structure de données requise pour filtrer l'automate consomme plus de mémoire que la méthode GAC. Touts les 500 retours arrières on mesure la quantité de mémoire utilisée par les trois approches. On donne ici la consommation moyenne en méga-octets pour les trois différentes version du filtrage.

instance	$Aut0$	$Aut1$	$Aut2$	GAC
15.07(dico :words)	31.8	21.1	16.1	19.7
19.02(dico :words)	37.2	25.2	17.6	19.1
21.03(dico :words)	51.6	33.9	22.2	23.7

Parmi les trois version, le meilleur compromis pour la consommation de temps et d'espace semble être $Aut1$. Cependant $Aut2$, qui est proche du GAC

sur le critère de la vitesse, mais qui requière moins de mémoire que l'algorithme GAC.

6.5.2 Enregistrement des nogoods

6.5.2.1 Crossword puzzles

Les puzzles de mots croisés ont été utilisé par Ginsberg [Gin93] pour démontrer l'efficacité du retour-arrière dynamique. Il s'agit de notre seconde raison pour regarder en particulier les mots-croisés comme nous travaillons sur les techniques d'enregistrement de nogoods. Les puzzles utilisés par Ginsberg sont des problèmes très structurés dans lesquels quelques affectations permettent de séparer le problème en parties indépendants. Le thrashing est commun dans ce genre de situation.

Résolution : Nous avons étudié trois approches :

1. MAC-CBJ utilise *aut*1 pour réaliser le filtrage. MAC-CBJ [Pro95] est une technique de retour-arriére intelligent qui consiste à backtracker à la dernière décision impliqué dans le conflit quand un échec survient.

2. MAC-CBJ et l'algorithme de Watched literals pour réaliser le filtrage en utilisant les nogoods.

3. MAC-CBJ et l'algorithme de Watched liteaux pour réaliser le filtrage en un utilisant les *nogoods* généralisés.

Résolution Nous avons utilisé les mêmes benchmarks mais une limite de temps fixé à 2 heures.

	MAC-CBJ		MAC-CBJ + S		MAC-CBJ + G	
	tps (s)	**n**ode	**t**ps (s)	**n**ode	**t**ps (s)	**n**ode
15.02(dico :words)			47,1	314	**4**5,5	**3**03
15.07(dico :words)			1326,9	17975	**9**28,3	**1**1220
19.02(dico :words)			16,5	264	**1**4,6	**2**19
19.05(dico :words)			69,1	1210	**4**7,8	**7**13
21.03(dico :words)			19,1	292	**1**8,5	**2**81
21.06(dico :words)			276,4	2677	**1**71,7	**1**878
21.07(dico :words)			75,7	1168	**6**5,9	**9**57
21.04(dico :words)			> 2h	68249	> 2h	50387
23.07(dico :words)			78,1	892	**6**3,9	**5**81
21.05(dico :words)			> 2h	92836	**7**62,8	**9**392
21.10(dico :words)			> 2h	45205	> 2h	42086
15.04(dico :words)			> 2h	100160	**1**340,2	**1**6718
15.06(dico :words)			> 2h	90175	**4**578,9	**4**4630
15.10(dico :words)			> 2h	96673	**1**642,5	**1**3990
19.03(dico :words)			> 2h	85063	> 2h	57492
19.04(dico :words)			> 2h	202381	**6**0,6	**2**433
19.07(dico :words)			> 2h	276985	**1**29,2	**4**394
21.01(dico :words)			> 2h	52786	> 2h	36262
23.03(dico :words)			> 2h	48538	> 2h	43120
23.04(dico :words)			> 2h	35774	> 2h	24607
23.05(dico :words)			17,1	254	**1**5,5	**2**26

Résultats : L'utilisation des Watched Literals implique un léger surcoût du au filtrage et à la gestion des nogoods. Mais l'utilisation des *nogoods* et en particulier des *nogoods* généralisés permet de visiter moins de nœuds et de résoudre le problème plus rapidement. Les *nogoods* généralisés permettent de résoudre six problèmes qui n'était pas résolus après deux heures de calculs lorsque les *nogoods* standards étaient utilisés.

6.5.2.2 Problèmes d'allocation de fréquences

Notre dernière expérience est composées de problèmes réels d'allocation de fréquences [CdGL$^+$99] issus de l'archive FullRLFAP. Le problème consiste à trouver des fréquences (f_i) pour différents canaux de communication minimisant les interférences. Les interférences sont exprimées en utilisant la contrainte de distance minimale entre les fréquences des différents canaux ($|f_i - f_j| > E_{ij}$ or $|f_i - f_j| = E_{ij}$). On se base sur [RD01, GW06] pour générer instance de satisfaction difficiles. Ainsi, `scenXX-wY-fZ` correspond à l'instance original `scenXX` où les contraintes avec un poids supérieur à `Y` sont retirés ainsi que les plus hautes fréquences `Z`.

	MAC-Cbj		MAC-Cbj + S		MAC-Cbj + G	
	tps (s)	node	tps (s)	node	tps (s)	node
scen02_5_24			0,9	104	0,4	104
scen02_5_25			5,2	610	**3**,1	**3**60
scen03_5_10			> 2h	339356	**1**35,1	**1**1575
scen03_5_11			> 2h	**7**69578	> 2h	152830
scen11_5_0			8,1	1207	**3**,8	**6**22
scen06_2_0			165,9	62655	**5**,5	**1**854
scen07_1_4			0,2	202	0,2	**2**01
scen07_1_5			0,1	26	0,1	26
graph08_5_10			> 2h	511219		
graph08_5_11						
graph14_5_27						

Résultats : Les problèmes RLFAP mènent aux mêmes conclusions que les puzzles de mot-croisés. Les *nogoods* généralisés permettent de visiter moins de nœuds et de résoudre le problème plus rapidement. Ces résultats prouvent l'intérêt des *nogoods* généralises en relation avec les *nogoods* standards pour le filtrage.

6.6 Conclusion

Nous avons présenté une méthode pour stocker dynamiquement les *nogoods* en utilisant un automate. On explique aussi comment gérer l'automate et comment l'utiliser pour réaliser le filtrage et calculer les explications. Nos expérimentations valident le fait que les automates sont un moyen efficace pour réaliser la GAC sur un grand nombre de tuples. De plus nous avons montré que le filtrage utilisant les *nogoods* généralises permet de résoudre les problèmes structures plus rapidement. Cependant, on ne peut pas actuellement gérer de manière dynamique les *nogoods* généralisé parce que la taille de l'automate ne permet pas d'effectuer le retrait et la minimisation rapidement. Aussi le travail futur va consister à trouver des heuristiques pour sélectionner et ordonner des sous-ensembles de *nogoods* qui peuvent être stockés efficacement dans un automate.

Par rapport à la problématique des problèmes dynamiques, l'utilisation des automates pour le stockage des nogoods permet de filtrer efficacement les parties de l'espace déjà explorées. De plus la gestion dynamique de l'ajout/retrait de tuples à la contrainte permet de s'adapter aux évolutions du modèle (ajout/suppression de contraintes) et leurs répercussions sur l'ensemble des nogoods.

Chapitre 7

Contraintes dynamiques

7.1 Dynamicité et contraintes monotones

Dans un CSP, le problème que l'on cherche à résoudre est modélisé à l'aide d'un ensemble de variables (correspondant aux différents objets du problème) et d'un ensemble de contraintes (correspondant aux relations reliant les objets entre eux).

Le retrait et l'ajout d'objets dans le problème se traduit par le retrait et l'ajout de variables dans le modèle. Généralement, cela se traduit aussi par une modification de l'ensemble des contraintes. Cependant, bien que l'ensemble des variables évolue, le problème modélisé reste sémantiquement identique : les relations entre les variables restent les mêmes et les variables représentent toujours les mêmes objets.

Ainsi dans le cadre des DCSP, un retrait de variable se traduira en pratique par le retrait des contraintes portant sur cette variable, puis par l'ajout de contraintes équivalentes auxquelles on aura retiré la variable supprimée. Cela est particulièrement lorsqu'on utilise des contraintes globales qui portent sur un grand nombre de variables du problème.

L'objectif de la réutilisation de contraintes est de mettre en place des techniques permettant de prendre en compte cette stabilité dans la sémantique du problème pour limiter le travail effectué lors de l'évolution du modèle. Nous nous sommes intéressés aux propriétés qui peuvent être exploitées dans le cadre dynamique : la monotonie et l'incrémentalité des contraintes.

7.1.1 Ajout de contraintes monotones au modèle

Dans [Bar01] l'auteur montre l'intérêt de l'ajout dynamique de variables à l'aide d'un exemple de planification utilisant des contraintes de ressources dans lequel des activités apparaissent durant la résolution.

Ce problème peut être résolu à l'aide d'un modèle statique auquel on ajoute des variables vides qu'on utilisera au fur et à mesure pour modéliser les nouvelles activités qui apparaissent. L'inconvénient de cette méthode est que les performances sont détériorées lorsque le nombre de variables à ajouter est trop important puisqu'elle implique qu'il y ait autant de variables vides que de variables à ajouter durant la recherche.

L'alternative proposée dans [Bar01, Bar03] est de rendre les contraintes dynamiques c'est à dire développer des contraintes auxquelles on peut ajouter et retirer dynamiquement des variables sans avoir à retirer effectivement les contraintes du réseau. L'utilisateur n'a pas besoin de connaitre le nombre de variables qui va être ajouté lors de la définition du problème. Et le modèle ne comporte que des variables utiles pour la propagation.

La méthode proposé par [Bar01, Bar03] repose sur la monotonie des contraintes (définition 17) utilisées pour modéliser le problème. Une contrainte monotone est une contrainte à laquelle on peut ajouter des variables sans ajouter de nouvelle solution. Par exemple, la contrainte $alldifferent$ est monotone

puisque toute sous-partie d'une solution est aussi solution (les éléments sont deux à deux distincts). À l'inverse, la contrainte *atleast* n'est pas monotone. En effet $atleast(2, 1, \{1, 0, 1\})$ est consistant alors que $atleast(2, 1, \{1, 0\})$ est inconsistant.

Définition 16. Soit une contrainte globale G portant sur l'ensemble des variables X notée $G(X)$. On dit que $G(Y)$ est une extension de $G(X)$ si $X \subseteq Y$

Définition 17. Soit $Sol(G(X), D(X))$ un ensemble des solutions de la contrainte globale G portant sur l'ensemble de variables X de domaine $D(X)$. La contrainte globale G est dite monotone si pour n'importe quel Y la propriété $Sol(G(X \cup Y), D(X \cup Y)) \downarrow X \subseteq Sol(G(X), D(X))$ est vérifiée.

Ainsi, on peut mettre en place une technique de dynamisation générique des contraintes monotones en utilisant le fait que les affectations inconsistantes pour la contrainte initiale le sont encore pour son extension.

Pour ajouter la variable X_{k+1} à la contrainte monotone $G(X_1, \ldots, X_k)$ avec $g(X_1, \ldots, X_k)$ une instance de G il faut :

1. désactiver $g(X_1, \ldots, X_k)$ c'est à dire, sauvegarder sur la pile les structures de données internes et ne plus utiliser cette contrainte pour la propagation
2. sauvegarder sur la pile les domaines de $X_1, \ldots, X_k, X_{k+1}$
3. ajouter la contrainte $g(X_1, \ldots, X_k, X_{k+1})$

Le retrait de variable dans le cas de contraintes monotones est soumis à de sévères limitations. En effet l'approche proposée par [Bar01, Bar03] ne permet de retirer uniquement les variables ajoutées dynamiquement et le retrait ne peut avoir lieu que lors d'un backtrack au monde dans lequel elles avaient été ajouté. Si ces conditions sont vérifiées, on peut retirer une variable X_{k+1} de la manière suivante :

1. désactiver et effacer $g(X_1, \ldots, X_k, X_{k+1})$
2. restaurer à partir de la pile les domaines de $X_1, \ldots, X_k, X_{k+1}$
3. activer $g(X_1, \ldots, X_k)$

Ainsi le retrait est géré par un mécanisme de backtrack standard.

On peut généraliser cette technique au cas des contraintes globales monotones par retrait (définition **??**). La monotonie par retrait nous permet de gérer de manière transparente le cas du retrait d'une variable.

Définition 18. Soit une contrainte globale G portant sur l'ensemble des variables X notée $G(X)$. On dit que $G(Y)$ est une réduction de $G(X)$ si $X \subseteq Y$.

Définition 19. Soit $Sol(G(X), D(X))$ un ensemble des solutions de la contrainte globale G portant sur l'ensemble de variables X de domaine $D(X)$. La contrainte globale G est dite monotone si pour n'importe quel Y la propriété $Sol(G(X), D(X)) \downarrow Y \subseteq Sol(G(Y), D(Y))$ est vérifiée.

Les contraintes globales qui sont monotones lors du retrait peuvent utiliser le même algorithme que celui utilisé par les contraintes globales monotones lors de l'ajout d'une variable.

1. désactiver $g(X_1, \ldots, X_k)$ c'est à dire, sauvegarder sur la pile les structures de données internes et ne plus utiliser cette contrainte pour la propagation
2. sauvegarder sur la pile les domaines de $X_1, \ldots, X_k$
3. ajouter la contrainte $g(X_1, \ldots, X_{k-1})$

Pour retirer une variable, on peut adapter la méthode proposée. Il faut alors commencer par désactiver les contraintes auxquelles on veut retirer la variable puis sauvegarder l'état interne des contraintes désactivées ainsi que les domaines des valeurs sur lesquelles elles portent. Enfin, on ajoute les contraintes réduites aux variables restantes. Inversement, pour ajouter une variable : on retire la contrainte précédemment ajoutée, on restaure les domaines et structures sauvegardés puis on réactive les contraintes.

Les restrictions de la technique proposée par [Bar01, Bar03] sont assez fortes. En effet, elle permet de gérer uniquement l'ajout de variables à des contraintes globales monotones (définiton 17) durant la recherche. Le retrait de variables se limite aux variables ajoutées et le retrait ne peut avoir lieu que lors d'un backtrack au monde dans lequel l'ajout a été effectué. De plus, elle implique que le solveur utilisé permettent l'ajout de contraintes au réseau et la désactivation de contraintes durant la recherche. Cependant lorsque les conditions sont réunies cette méthode est relativement efficace puisqu'elle permet d'ajouter ou retirer une variable avec pour seul surcoût celui engendré par la sauvegarde des structures internes de la contrainte et par la sauvegarde des domaines des variables.

En plus de ces limitations cette technique pose des problèmes d'efficacité. En effet, le stockage des structures internes des contraintes globales après chaque ajout de variable peut être très couteux du point de vue de la consommation de mémoire. L'autre source d'inefficacité est dûe au fait que pour ajouter une nouvelle variable il faut réinitialiser complètement les structures internes utilisées par la contrainte. Afin d'éliminer ces deux sources d'inefficacité [Bar01, Bar03] propose de développer des contraintes dynamiques utilisant des structures incrementales (qui permettent l'ajout de variables et le retrait lors du backtrack) sans avoir à réinitialiser les structures de données.

Il explique alors comment maintenir la contrainte $alldiff$ lors de l'ajout d'une variable, le retrait étant géré grâce au mécanisme de backtrack standard. [Bar03] s'appuie sur l'implémentation de la contrainte $alldiff$ proposée par [Ré94]. L'algorithme de filtrage employé par [Ré94] repose sur le calcul de couplages maximum dans un graphe biparti variables/valeurs. Les arcs n'appartenant à aucun couplage maximum sont supprimés (retirer un arc du graphe valeur est équivalent à retirer une valeur du domaine de la variable). De plus

si le couplage maximum ne couvre pas toutes les variables des la contrainte $alldiff$, la contrainte échoue.

La contrainte $alldiff$ est monotone. Aussi lorsque nous ajoutons une nouvelle variable nous n'introduisons pas de nouvelles solutions. Ainsi l'extension de l'ensemble des variables sur lesquelles porte la contrainte ne remet pas en cause les décisions antérieures. Les arcs supprimés, car n'appartenant à aucun couplage maximum, n'appartiennent à aucun couplage maximum dans un graphe valeurs étendu. On peut ainsi étendre de manière incrémentale le graphe valeurs en ajoutant au graphe initial des arcs reliant les nœuds correspondant aux nouvelles variables aux nœuds correspondant aux valeurs. Il n'y a pas de nouveaux arcs reliant une anciennes variables à une valeur (les domaines des anciennes variables restent inchangés). Le nouveau couplage maximum est ensuite calculé à partir de l'ancien.

	Space	Time
Dynamisation générique - AddVariable(AllDiff)	$O(dp)$	$O(dp\sqrt{p})$
Alldifferent dynamique - AllDiff-Update	$O(d+p)$	$O(dp)$

FIGURE 7.1: Comparaison de la complexité spatiale et temporelle de l'ajout d'une variable pour la dynamisation générique et pour l'implémentation de la contrainte $alldifferent$. p est le nombre de variables dans la contrainte et d le nombre de valeurs dans le domaine de ses variables

Le surcoût en espace de la dynamisation générique est dû au fait que pour introduire une nouvelle variable dans la contrainte $alldiff$ nous devons sauvegarder l'intégralité de sa structure interne (graphe valeurs biparti). Le surcoût en temps, vient du fait que la dynamisation générique implique de reinitialiser les structures de données à chaque nouvel ajout ($O(dp\sqrt{p})$) alors que le $alldiff$ se limite à étendre le couplage maximum du graphe valeurs biparti.

Jusqu'à présent nous nous sommes concentrés sur le cas des contraintes monotones. Lorsque l'on considère des contraintes non monotonzs la difficulté est que l'ajout de nouvelles variables peut entraîner l'apparition de nouvelles solutions.

Ainsi si on reprend l'exemple de [Bar03] : on considère trois variables x_1, x_2 et x_3 de domaine $\{1,2,3\}$ et la contrainte $sum(x_1,x_2,x_3,9)$ qui impose que la somme des trois variables soit égale à 9. Les valeurs 1 et 2 peuvent être retirées des domaines puisqu'elles ne peuvent pas participer à une solution. Cependant, le filtrage n'est plus valide si on ajoute, à l'ensemble des variable, la variables x_4 de domaine $\{1,2,3\}$ et qu'on étend la contrainte sum pour prendre en compte la variable x_4 ($sum(x_1,x_2,x_3,x_4,9)$). En effet, l'affectation $x_1=1, x_2=2, x_3=3, x_4=3$ devient solution.

Une possibilité pour utiliser une contrainte non monotone est d'attendre que toutes les variables aient été découvertes avant de la poser. Cela contourne

la limitation mais pose des grosses difficultés pour l'efficacité du filtrage puisqu'on traite alors un problème relaxé.

La seconde possibilité proposée par [Bar03] est de rendre dynamique les contraintes en permettant l'ajout et le retrait incrémental de valeurs du domaine. En effet, lorsqu'une nouvelle variable est ajoutée à une contrainte non-monotone, elle peut remettre en cause des suppressions antérieures de valeurs. Nous devons donc pouvoir restaurer les domaines des variables remises en cause soit en réintroduisant toutes les valeurs supprimées (ce qui peut engendrer un surcoût important), soit en restaurant incrémentalement les valeurs. Cela implique d'utiliser un solveur de contraintes autorisant la gestion incrémental des contraintes.

La dernière solution proposée est d'approximer les contraintes non-monotones par des contraintes monotones. La contrainte monotone permet alors d'utiliser les techniques décrites précédemment. Une fois que l'ensemble des variables est fixe, la version non monotone de la contrainte peut être utilisée. L'avantage est double : cette technique peut être intégrée à un solveur classique et le filtrage peut être réalisé dès le début de la résolution.

Nous nous intéresserons à la seconde possibilité proposée en approfondissant la question de l'incrémentalité des structures utilisées par les contraintes globales.

Détéction de la monotonie L'exemple des contraintes monotones nous a montré qu'on peut exploiter les propriétés des contraintes pour faciliter la gestion du dynamisme. Nous nous sommes donc intéressés au moyen d'identifier les propriétés des contraintes utilisées, au travers de l'exemple de la monotonie.

Avec la représentation à l'aide d'un graphe, une contrainte globale est représentée comme un digraphe dans lequel chaque nœud correspond à une variable et chaque arc correspond à une contrainte binaire entre les deux variables. Le graphe dépend de la contrainte à laquelle on s'intéresse. La principale différence avec un réseau de contraintes classique est dûe au fait qu'on ne force pas tous les arcs à être vérifiés. On considère le graphe duquel on a retiré l'ensemble des contraintes qui ne sont plus vérifiées ainsi que l'ensemble des nœuds isolés et on impose des propriétés de graphe sur ce graphe restant. Ces propriétés peuvent être des restrictions sur le nombre de composantes fortement connexes, sur la taille de la plus petite composante fortement connexe ou sur la taille de la plus grande composante fortement connexe. Ainsi les contraintes globales peuvent être décrites par l'intermédiaire des propriétés de graphe [NMST07].

Pour chaque contrainte globale on peut générer un graphe correspondant (prenant en compte les paramètres de la contrainte). Si on prend l'exemple des contraintes $alldifferent$, $atmost$, $atleast$ et $tree$:

– $alldifferent(VARIABLES)$ oblige les variables à prendre des valeurs deux à deux distinctes (figure 7.2). Le graphe correspondant possède

un nœud par variable de $VARIABLES$, deux nœuds sont reliés par un arc si et seulement si les deux variables correspondantes possèdent une même valeur dans leurs domaines. Il vérifie la propriété de graphe $MAX\_NSCC \leq 1$ où MAX_NSCC correspond au nombre de nœuds dans la plus grande composante fortement connexe

- $atmost(N, VARIABLES, VALUE)$ interdit à plus de n variables de prendre une valeur donnée (figure 7.4). Le graphe correspondant possède un nœud par variable. Les nœuds correspondant à une variable qui a la valeur dans son domaine ont une boucle sur eux mêmes. Ils vérifient la propriété de graphe $NARC \leq N$ avec $NARC$ le nombre d'arcs présents dans le graphe
- $atleast(N, VARIABLES, VALUE)$ oblige au moins n variables à prendre une valeur donnée (figure 7.3). Le graphe est généré de la même manière mais il vérifie la propriété de graphe $NARC \geq N$ avec $NARC$ le nombre d'arcs présents dans le graphe
- $tree(NTREES, NODES)$ qui tente de couvrir un graphe orienté $\mathcal{G}$ (définit par $NODES$) par un ensemble d'arbres nœuds disjoints (figure 7.5). $NTREES$ est une variable indiquant le nombre d'arbre à utiliser. Le graphe correspond à celui défini par $NODES$. Il doit vérifier les propriétés $MAX\_NSCC \leq 1$ et $NCC = NTREES$ avec NCC le nombre de composantes connexes dans le graphe.

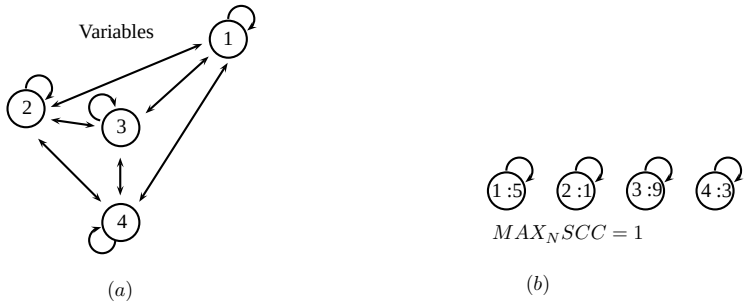

FIGURE 7.2: Exemple $alldifferent(\langle 5, 1, 9, 3\rangle)$. Description de la contrainte $alldifferent$ à l'aide des propriétés de graphe. Le graphe initial est construit en créant un nœud par variables et en ajoutant un arc entre chaque nœuds dont les variables correspondantes possèdent la même valeur dans leur domaine. L'invariant utilisé est la taille de la plus grande composante fortement connexe dans le graphe.

Soit V un ensemble de variables avec $Y \subseteq V$ et $X \subseteq V$. Dire qu'une contrainte C est monotone signifie que :

$$\forall Y \subseteq X, x \in Sol(G(X)) \Rightarrow x \downarrow Y \in Sol(G(Y))$$

FIGURE 7.3: Exemple $atleast(1, \langle 4, 2, 4, 5\rangle, 2)$. Description de la contrainte *atleast* à l'aide des propriétés de graphe. Le graphe initial est construit en créant un nœud par variables et en ajoutant une boucle sur chaque nœud dont la variable correspondante possède la valeur $VALUE$ dans son domaine. L'invariant utilisé est le nombre d'arcs présents dans le graphe.

FIGURE 7.4: Exemple $atmost(1, \langle 4, 2, 4, 5\rangle, 2)$. Description de la contrainte *atmost* à l'aide des propriétés de graphe. Le graphe initial est construit en créant un nœud par variables et en ajoutant une boucle sur chaque nœud dont la variable correspondante possède la valeur $VALUE$ dans son domaine. L'invariant utilisé est le nombre d'arc présent dans le graphe.

Vérifier si une contrainte est monotone revient donc à trouver la transformation qui permet de passer du graphe $\mathcal{G}_n$ représentant la contrainte posée sur n variables au graphe $\mathcal{G}_{n-1}$ représentant la contrainte posée sur $(n-1)$ variables et de vérifier : si les propriétés de graphe sont vérifiées pour $\mathcal{G}_n$ alors elles le sont pour $\mathcal{G}_{n-1}$.

Inversement, dire qu'une contrainte C est monotone en retrait signifie :

$$\forall Y, X \subseteq Y, x \in Sol(G(X)) \Rightarrow x \in Sol(G(Y)) \downarrow Y$$

C'est à dire, vérifier que si les propriétés sont vérifiées pour $\mathcal{G}_{n-1}$ alors elles le sont aussi pour le graphe $\mathcal{G}_n$. Ainsi si on prend le cas des contraintes présentées plus haut (figure 7.6, figure 7.7, figure 7.8 et figure 7.9). On peut donc déterminer la monotonie d'une contrainte à partir de son graphe. L'autre avantage du graphe c'est qu'il permet d'identifier des sous-cas où la contraintes est monotone. Par exemple, ajouter ou retirer une variable correspondant à un nœud ne possédant que des arcs sortant ne rajoute pas de solution (dans une solution ce nœud sera forcement une feuille).

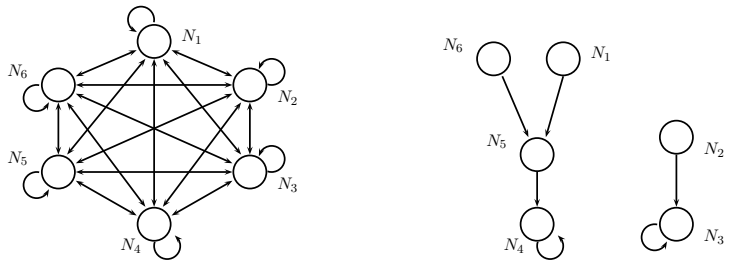

FIGURE 7.5: La contrainte `tree` à pour but de couvrir un graphe orienté G par un ensemble d'arbres, de telle sorte que chaque nœud de G appartienne à un unique arbre. Les arcs des arbres sont dirigés des feuilles vers leur racine respective. Le graphe initial est construit en créant un nœud n_i par variable v_i, et en créant des arcs (n_i, n_j) lorsque $j \in dom(v_i)$. Le graphe final doit vérifier des propriétés de graphe portant sur le nombre de composante connexe (NCC) et la taille de la plus grande composante fortement connexe (MAX_NSCC) : $NCC = NTREES$ et $MAX\_NSCC \leq 1$.

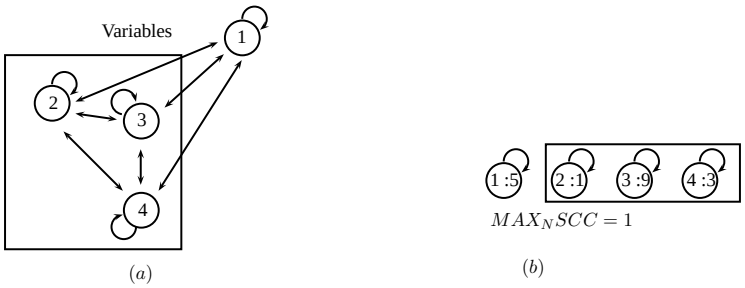

FIGURE 7.6: Transformer le graphe $\mathcal{G}_n$ en graphe $\mathcal{G}_{n-1}$ consiste à retirer un sommet du graphe. Ainsi la propriété $MAX\_NSCC = 1$ est toujours vérifié pour le graphe $\mathcal{G}_{n-1}$ si elle l'était pour le graphe $\mathcal{G}_n$.

L'utilisation de structures incrémentales est une solution proposée par [Bar01, Bar03] pour permettre l'ajout/retrait de variables, dans le cas de contraintes non monotones. Nous nous sommes donc intéressés à la mise en place de contraintes globales incrementales.

7.2 Incrémentalité et maintien des structures

Avec l'implémentation de la contrainte $alldiff$, proposée dans [Bar01, Bar03], il apparaît que l'utilisation de contraintes dynamiques disposant d'algorithme de maintien incrémentai des structures internes peut être une solution intéressante pour gérer les CSP dynamiques. L'incrémentalité semble particulièrement indiquée dans le cadre des problèmes dynamiques car elle permet d'effectuer des recherches non arborescentes et de limiter les recal-

FIGURE 7.7: Transformer le graphe $\mathcal{G}_n$ en graphe $\mathcal{G}_{n-1}$ consiste à retirer un sommet du graphe. Ainsi la propriété $NARC \geq Iter$ n'est plus vérifiée lorsque que pour $\mathcal{G}_n$ on a $NARC = Iter$ et que l'on retire une variable qui était instancié a $VALUE$.

FIGURE 7.8: Transformer le graphe $\mathcal{G}_n$ en graphe $\mathcal{G}_{n-1}$ consiste à retirer un sommet du graphe. Ainsi la propriété $NARC \leq Iter$ est forcement vérifier pour $\mathcal{G}_n$ si elle l'est pour $\mathcal{G}_{n-1}$.

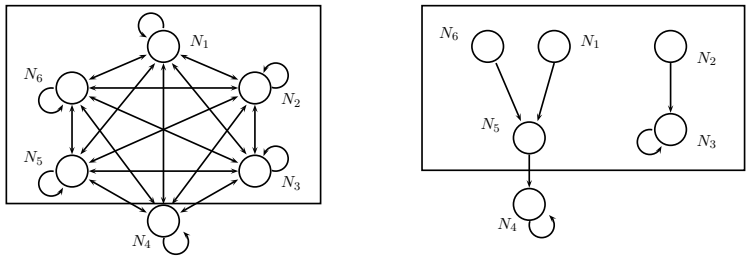

FIGURE 7.9: Transformer le graphe $\mathcal{G}_n$ en graphe $\mathcal{G}_{n-1}$ consiste à retirer un sommet du graphe. Ainsi la propriété $MAX\_NSCC \leq 1$ n'est pas forcement vérifier pour $\mathcal{G}_n$ si elle l'est pour $\mathcal{G}_{n-1}$.

culs lors de la modification des structures de données. Cependant l'approche proposée souffre toujours des limitations exposées plus haut : l'ajout d'une variable se limite au cas des contraintes monotones et le retrait n'est possible que lors du backtrack. Nous avons donc tenté de généraliser l'utilisation des structures incrementales dans le cadre des contraintes globales et de développer des contraintes indépendantes des mécanismes de backtrack fournit par le solveur. Pour illustrer notre propos, nous nous sommes intéressés à la contrainte *tree*.

7.2.1 La contrainte `tree`

La contrainte *tree* permet de partitionner un graphe orienté en une forêt d'arbres disjoints [BFL05b]. Plus précisément, le graphe est partitionné en un ensemble d'anti-arbres n'ayant aucun nœud en commun. La contrainte *tree* permet de modéliser de nombreux problèmes liés aux graphes comme les problèmes d'arbre phylogénétique, les problèmes de chemins orientés ou les problèmes de planification de missions.

Une instance de la contrainte `tree` est représentée par un graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ dans lequel les nœuds représentent les variables, $\mathcal{V} = \{v_1, \ldots, v_n\}$, les arcs représentent la relation de succession directe entre eux, $dom(v_i) = \{j \mid (v_i, v_j) \in \mathcal{E}\}$, et `ntree` est une variable qui spécifie le nombre d'arbres dans la forêt ($\underline{\texttt{ntree}}$ et $\overline{\texttt{ntree}}$ correspondent respectivement à la valeur minimal et à la valeur maximale de $dom(\texttt{ntree})$). Une instance de la contrainte $\texttt{tree}(\texttt{ntree}, \mathcal{G})$ spécifie que le graphe orienté associé $\mathcal{G}$ devrait être une forêt de `ntree` arbres, formellement :

Définition 20. Une *instance solution* de la contrainte $\texttt{tree}(\texttt{ntree}, \mathcal{G})$ est une *solution* si et seulement si (1) le graphe orienté associé $\mathcal{G}$ est composé de `ntree` composantes connexes, et (2) chaque composante connexe de $\mathcal{G}$ n'a pas de circuit impliquant plus d'un nœud (notons que chaque composante contient exactement un nœud possédant une boucle et qui correspond à la racine de l'arbre).

Nous rappelons quelques définitions et notations portant sur le graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.

Définition 21. Graphe réduit À chaque instance de la contrainte $\texttt{tree}(\texttt{ntree}, \mathcal{G})$ on associe un graphe réduit $\mathcal{G}_r$ issue de $\mathcal{G}$ de la manière suivante : à chaque composante fortement connexe de $\mathcal{G}$ on associe un nœud de $\mathcal{G}_r$; à chaque arc de $\mathcal{G}$ reliant deux composantes fortement connexe différentes correspond un arc dans $\mathcal{G}_r$.

Notation 1. ***Composante puits*** *Une composante fortement connexe de $\mathcal{G}$ qui correspond à un puits de $\mathcal{G}_r$ est appelée composante puits.*

Notation 2. ***Racine potentielle*** *Un nœud v de $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ tel que $(v, v) \in \mathcal{E}$ est appelé racine potentielle et l'arc (v, v) est appelé boucle.*

Définition 22. Dominateur Étant donné un graphe orienté $\mathcal{G}$ et deux nœuds distincts i, j de $\mathcal{G}$ tels qu'il existe au moins un chemin allant de i à j, un nœud d est un dominateur de j par rapport à i si et seulement si il n'y a aucun chemin allant de i à j dans $\mathcal{G} \subset \{d\}$. L'ensemble des dominateurs de j par rapport à i est noté $DOM_{<\mathcal{G},i>}(j)$.

Étant donné un graphe orienté $\mathcal{G}$ et un ensemble de *racines potentielles* (notation 2), le nombre minimal d'arbres ($\underline{\texttt{ntree}}$) pour partitionner le graphe

orienté $\mathcal{G}$ associé à une contrainte `tree` est le nombre de puits (c.-à-d. de nœuds sans arc sortant exceptée la boucle sur lui-même) du graphe réduit (définition 21) de $\mathcal{G}$, et le nombre maximum d'arbres ($\overline{\texttt{ntree}}$) pour partitionner le graphe orienté $\mathcal{G}$ est le nombre de racines potentielles.

7.2.1.1 Filtrage

Ensuite, nous détaillons l'algorithme de filtrage de la contrainte `tree`. L'algorithme se présente sous la forme d'un ensemble de règles dépendantes de la variable `ntree` et des *noeuds dominants*.

Lorsque la variable `ntree` doit atteindre la valeur $\overline{\texttt{ntree}}$, l'algorithme force, pour chaque racine potentielle, l'arc boucle (qui représente le fait qu'il s'agit d'une racine potentielle). Dans le cas où `ntree` doit atteindre $\underline{\texttt{ntree}}$, l'algorithme retire, pour chaque *racine potentielle* qui n'appartient pas à une composante fortement connexe puits de $\mathcal{G}$, la boucle sur lui-même.

Proposition 1. *Le domaine de la variable $NTREE$ est limité par :*

1. *Si $max(NTREE) > MAXTREE$ alors $max(NTREE) = MAXTREE$*
2. *Si $min(NTREE) < MINTREE$ alors $min(NTREE) = MINTREE$*

Finalement, pour n'importe quel `ntree` la règle principale de filtrage associée avec la contrainte est basée sur la détection des *noeuds dominants* du graphe (définition 22). Alors, l'algorithme de filtrage doit détecter tous les nœuds de j de $\mathcal{G}$ tel qu'il existe un nœud i pour lequel j domine toutes les racines potentielles de $\mathcal{G}$ par rapport à i. Les arcs infaisables dans $\mathcal{G}$ pour une contrainte `tree` sont les arcs sortant (j, k), où j est un nœud dominant, tel qu'il n'existe pas un chemin de i à une racine potentielle de $\mathcal{G}$ utilisant l'arc (j, k). En effet si un arcs sortant (j, k) est utilisé alors il n'existe plus de chemin permettant de relier i à une racine potentielle.

Proposition 2. *Soit $\mathcal{C} = dom(NTREE) \bigcap [MINTREE, MAXTREE]$. Pour chaque composante fortement connexe $\mathcal{S}$ of $\mathcal{G}$:*

1. *Si $\mathcal{S}$ est une composante puits de $\mathcal{G}$ qui contient une unique racine potentielle r, alors les arcs sortant de r, autres que l'arc (r, r) sont infaisables.*
2. *Sinon*
   a) *Si $\mathcal{C} = \{MAXTREE\}$ alors, pour chaque racine potentielle r de $\mathcal{S}$, tous les arcs (r, v) avec $v \neq r$ sont infaisables*
   b) *Si $\mathcal{C} = \{MINTREE\}$ et $\mathcal{S}$ n'est pas une composante puits alors tout les boucles de $\mathcal{S}$ sont infaisable.*
   c) *Si il existe un unique nœud winner w dans $\mathcal{S}$, qui est une door, alors tout les non-connecting arcs (w, v) sont infaisables*

Initial awakening of the `tree` constraint :
- Compute $\underline{\texttt{ntree}}$ and $\overline{\texttt{ntree}}$.
- If there is at least one solution satisfying the constraint then, do propagation related to the constraint :
  1. Update `ntree` according to $\underline{\texttt{ntree}}$ and $\overline{\texttt{ntree}}$.
  2. Propagate according to the dominator nodes of $\mathcal{G}$.
  3. Propagate according to the values of max(`ntree`) and min(`ntree`).

Each time an event occurs on a domain variable of the `tree` constraint do :
- If this event occurs on a domain variable modeling `ntree` then :
  1. Update `ntree` according to $\underline{\texttt{ntree}}$ and $\overline{\texttt{ntree}}$.
  2. Propagate according to max(`ntree`) and min(`ntree`).
- If this event occurs on a domain variable modeling a node of $\mathcal{G}$ do :
  1. Update `ntree` according to $\underline{\texttt{ntree}}$ and $\overline{\texttt{ntree}}$.
  2. Propagate according to new dominator nodes of $\mathcal{G}$.

FIGURE 7.10: Squelette d'implémentation de la contrainte `tree` au sein de *choco*.

La Figure 7.10 décrit le squelette de la contrainte `tree` telle qu'elle a été implémentée dans le solveur `choco`. Ici, les effets des événements survenant sur les variables du graphe consistent en plusieurs modifications de la structure du graphe. Par exemple, si un arc (i, j) est retiré de $\mathcal{G}$ (c.-à-d., $j \notin dom(v_i)$) alors, ce retrait peut : **(1)** diminuer le nombre de racines potentielles, si $i = j$. Cela mène à une mise à jour de $\overline{\texttt{ntree}}$. **(2)** augmenter le nombre de composantes puits. Cela mène à une augmentation de $\underline{\texttt{ntree}}$. **(3)** augmenter le nombre de composantes fortement connexes (cfc) dans $\mathcal{G}$. Cela mène à modifier le graphe réduit $\mathcal{G}_r$ associé avec $\mathcal{G}$. **(4)** créer un nouveau nœud dominant dans $\mathcal{G}$.

Le filtrage repose donc sur la recherche des nœuds dominants dans un graphe, la détection des composantes puits et des racines potentielles et le maintien du graphe réduit.

7.2.1.2 Implémentation

Nous allons maintenant détailler l'algorithme et l'implémentation utilisés dans la version statique de la contrainte *tree*. Nous proposerons ensuite une version dynamique utilisant des structures incrémentales.

Du point de vue de l'implémentation, une instance de la contrainte `tree` est composée d'une variable `ntree` dont le domaine $[\underline{\texttt{ntree}}, \overline{\texttt{ntree}}]$ correspond au nombre d'arbres autorisés dans la foret et un ensemble de variables $\mathcal{V}$ représentant le graphe $\mathcal{G}$ tel que à chaque nœud i de $\mathcal{G}$ correspond une variable $v_i \in mV$ et $j \in dom(v_i)$ si il existe un arc $(v_i, v_j) \in \mathcal{G}$. En interne `tree` utilise des structures de données pour maintenir et calculer les informations nécessaires pour réaliser le filtrage : la liste des prédécesseurs, la liste des successeurs de chaque nœuds, le graphe réduit $\mathcal{G}_\nabla$. Ces structures sont mises à jour après chaque modification des domaines des variables.

Si le retrait d'une valeur d'une variable conduit à vider un domaine, il

faut alors retourner dans un état consistant antérieur. Pour restaurer les informations nécessaires pour la contrainte nous utilisons les structures backtrackable fournie par le solveur de contraintes. L'implémentation originale de la contrainte *tree* utilisait ces structures afin de stocker et pouvoir enregistrer les propriétés de graphe comme les composantes fortement connexes et les nœuds dominateurs du graphe orienté $\mathcal{G}$.

Le solveur Choco propose plusieurs structures *backtrackables* utilisant des entiers, des booléens et des bitsets. Choco utilise le principe du *trailing* [Sch99] pour enregistrer les modifications (instantiations des variables, retraits des valeurs des domaines, etc) et ces effets sur les structures de données utilisées dans la contrainte. L'implémentation initiale de la contrainte `tree` utilisait ces structures de données fournies par le solveur afin d'enregistrer dynamiquement et restaurer les propriétés de graphe comme les composantes fortement connexes et les nœuds dominants du graphe orienté $\mathcal{G}$. Ainsi, l'utilisation de ces structures de données backtrackables dédiées fait que la contrainte `tree` initiale est dépendante du solveur.

Du point de vue du filtrage, l'implémentation initiale de la contrainte `tree` a été réalisé pour le solveur *Choco*. Choco est un solveur événementiel orienté variable, ce qui signifie que les contraintes connaissent les causes de leurs réveils (retrait de valeurs, modification des bornes, ...). Ainsi à chaque réveil la contrainte `tree` sait si le domaine de $NTREE$ a été modifié ou si des arcs du graphe (modélisé par les domaines des variables) ont été supprimés.

7.2.2 Implementation d'une contrainte `tree` incrémental

Le but de l'incrémentalité est de limiter les recalculs lors d'une modification atomique des données. Dans le cadre de la contrainte *tree* les modifications atomiques correspondent aux modifications du graphe la représentant : ajout d'un arc, retrait d'un arc. Ces deux modifications correspondent aux transformations de base qui peuvent survenir durant l'exploration de l'espace de recherche. L'ajout d'un arc correspond à l'ajout d'une valeur et le retrait d'un arc correspond à la suppression d'une valeur. Développer des algorithmes incrémentaux (qui savent gérer l'ajout et le retrait) pour maintenir les propriétés utilisées par le filtrage semble donc intéressant. Nous avons donc développé une contrainte *tree* utilisant des algorithmes incrémentaux. Comme expliqué dans [BFL05b], le filtrage réalisé par la contrainte *tree* repose sur les composantes connexes, les composantes fortement connexe et les nœuds dominants. Nous avons donc tenté de réduire au maximum les calculs nécessaires pour maintenir les propriétés lors d'une modification en identifiant les cas dans lesquels la perturbation n'a pas d'influence sur la propriété.

L'état de l'art classique des algorithmes de graphe propose plusieurs algorithmes incrémentaux permettant de maintenir les propriétés de graphe impliquées dans la contrainte `tree` comme les composantes fortement connexes et la fermeture transitive [EGI97], ou encore les nœuds dominants [SGL97].

Graph Order	Density	Average time (ms)	
		Ad-hoc	Pluggable
25	≤ 0.5	55	45
	> 0.5	90	38
50	≤ 0.5	610	310
	> 0.5	1532	307
75	≤ 0.5	3856	1174
	> 0.5	8896	1064
100	≤ 0.5	13040	3156
	> 0.5	32568	2682
150	≤ 0.5	69220	11543
	> 0.5	219174	9645
200	≤ 0.5	204497	33763
	> 0.5	> 300000	26315

TABLE 7.1: Comparaison des temps de résolution du problème du chemin hamiltonien à l'aide de contraintes classiques et de contraintes dynamiques (en ms).

Deux questions demeurent : est-il vraiment nécessaire d'utiliser des structures de données backtrackables lorsque des algorithmes incrémentaux en ajout et en retrait existent ? Si aucune structure backtrackable n'est utilisée (c.-à.-d. qu'il n'est pas nécessaire de *trailer* les modifications des structures de données utilisées dans la contrainte), quelles sont les relations entre les contraintes et le solveur ?

Nous avons réalisé plusieurs expérimentations afin de nous assurer de l'intérêt de l'incrémentalité dans les contraintes. Nous avons donc comparé une version de la contrainte `tree` utilisant les mécanismes de backtracks fournis par le solveur et une version reposant sur des structures incrémentales. Toutes les expérimentations ont été réalisées avec le solveur Choco sur un ordinateur possédant un processeur Intel Xeon 2,4GHz et 1Gb de mémoire vive.

Le but de ces expérimentations est de prouver qu'une version incrémentale de la contrainte `tree` peut être plus efficace que la version originale. De plus, nous avons mis en évidence que l'approche dynamique, en moyenne, est moins sensible aux variations de densité du graphe d'entrée. Pour chaque taille de graphe, dans $\{25, 50, 75, 100, 150, 200\}$ et les densités comprises dans $[0, 05; 1]$ avec un pas de 0,05, nous avons généré 30 instances (soit 3600 graphes au total). Une limite de temps est fixée à 300000ms et la recherche utilise un heuristique de choix de variable/valeur aléatoire.

Premièrement le tableau 7.1 met en évidence que la version dynamique est globalement plus efficace que la version originale. En effet, la version dynamique est 3,8 plus efficace pour des densités inférieures ou égales à 0,5 et 10 fois plus efficace pour des densités supérieures à 0,5. Les figures 7.11 et 7.12 montrent que la contrainte dynamique est significativement plus efficace

sur les graphes denses. En fait, la figure 7.11 décrit le comportement de la contrainte dynamique et de la contrainte de base pour des graphes de taille 100. La figure 7.12 donne le ratio entre les temps de la contrainte dynamique et de la contrainte d'origine. Dans les deux cas on observe que la version dynamique surpasse très nettement la version d'origine pour les graphes denses.

Cette différence de performance repose sur la manière différente dont sont calculées les composantes fortement connexes dans les deux versions de la contrainte. En effet dans la version originalle les composantes fortement connexes sont calculées en utilisant une recherche en profondeur d'abord, décrite par Tarjan [Tar72], de complexité $O(n+m)$. Or recalculer les composantes fortement connexes à chaque réveil de la contrainte est inutile. En effet durant les phases de propagations/recherches, retirer ou ajouter des arcs est une modification locale du graphe, on peut donc réduire ce coût en ne recalculant les composantes fortement connexes que sur le sous-graphe nécessaire. En pratique, durant la recherche, la taille des composantes diminue (ainsi que le nombre d'arcs) jusqu'à être égal à 1.

De plus, le filtrage requiert l'utilisation des nœuds dominants. Pour chaque nœud dominant, l'algorithme de filtrage détecte et retire les arcs sortants qui ne permettent pas d'atteindre au moins une racine potentielle. Ainsi pour chaque nœud dominant, on doit calculer un arbre de recherche en profondeur d'abord pour détecter si une racine potentielle peut être atteinte (complexité $O(mn)$). Dans la contrainte dynamique une nouvelle approche est utilisée. En plus du graphe, sa fermeture transitive est maintenue. La fermeture transitive est calculée une seule fois complètement lors de l'initialisation de la contrainte (complexité $O(mn)$), elle est ensuite maintenue incrémentalement. En pratique le maintient de la fermeture transitive est très efficace. L'information fournit par la fermeture transitive du graphe permet de savoir si un nœud est accessible ou pas et permet le filtrage dynamique des nœuds dominants lorsqu'ils sont identifiés par l'algorithme.

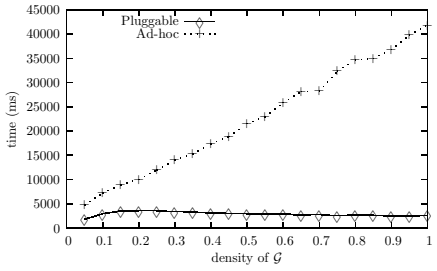

FIGURE 7.11: Évaluation

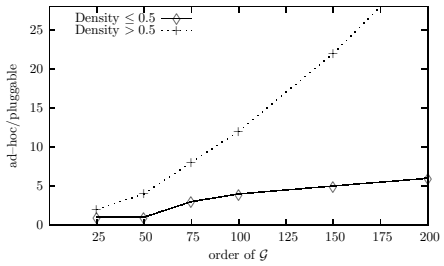

FIGURE 7.12: Évaluation de la contrainte `tree` portable et de l'implémentation *ad-hoc* pré-existante. À droite la courbe en pointillée indique le ratio entre les temps d'exécution de la contrainte *ad-hoc* et ceux de la contrainte portable, dans le cas de densité supérieure à $0,5$. L'autre courbe indique le ratio dans les cas de densité inférieure ou égale à $0,5$.

7.3 De l'incrémentalité à la généricité

Habituellement les contraintes utilisent les structures de données fournies par le solveur, ainsi le solveur gère seul les problèmes de restauration des structures lors d'un backtrack. En utilisant des structures de données incrémentales, on rend les contraintes indépendantes du solveur puisqu'on définit nos propres structures, ainsi que les méthodes pour les modifier et les restaurer. La deuxième source de dépendance des contraintes par rapport au solveur est dûes à la manière spécifique à chaque solveur d'interfacer une contrainte. Nous avons étudié l'intérêt que pouvait représenter le fait de développer des contraintes génériques, c'est à dire : des contraintes définissant leurs propres structures et utilisant une interface générique pour communiquer avec le solveur.

Passer d'une contrainte dépendante du solveur à une contrainte prête à brancher nécessite de spécifier comment la contrainte considérée va interagir avec ledit solveur. Cela est généralement assez simple. En effet, en pratique, les algorithmes de résolution des contraintes gèrent les réveils des contraintes selon les événements qui surviennent dans le domaine des variables sur lesquelles elles portent (dans le cadre des solveurs centrés événements), ou d'après des algorithmes généraux qui réalisent la propagation d'une manière donnée (pour les autres).

Dans le cadre des contraintes découplées des solveurs, les modifications dans les domaines des variables depuis le dernier réveil de la contrainte doivent être détectées et ensuite, traduites en événements gérables par la structure de données de la contrainte. De manière symétrique, nous avons à traduire les informations de filtrage de la contrainte en événements compréhensibles par le solveur (retraits de valeur, mise à jour des bornes, etc.) et applicables sur les domaines des variables qui peuvent être interprétées par l'algorithme de résolution basé sur les contraintes. Cette traduction bidirectionnelle des infor-

mations est appelée *interface* dans la suite. On peut remarquer qu'une interface peut être vue comme vision d'un niveau plus élevé des *advisors* introduits par [LS07], dans le sens où une interface s'occupe de diriger la stratégie des événements entre les contraintes et le solveur. Par conséquent, elle décide si une contrainte doit être propagée ou si le processus peut être retardé.

La transformation d'une contrainte globale en une contrainte prête à brancher devient complexe lorsqu'on considère des contraintes à états explicites. Pour résumer, une *contrainte à état explicite* gère sa propre structure de données afin de maintenir certaines propriétés qui sont utilisées par l'algorithme de filtrage pour retirer les valeurs inconsistantes dans les domaines des variables sur lesquelles la contrainte porte. Plus précisément, une contrainte globale s'appuie énormément sur les structures de données backtrackables du solveur hôte utilisé, et, dans le but de la rendre indépendante du solveur, nous devons gérer le backtrack dans la contrainte explicitement.

Par exemple, les contraintes arithmétiques binaires classiques ($\leq$, $=$, $\neq$) ne requièrent pas plus d'information que l'état du domaine de la variable. À l'opposé, les contraintes globales basées sur les graphes doivent modéliser une représentation du graphe dans le but de représenter efficacement et de manière expressive les propriétés liées (composantes connexes, composantes fortement connexes, nœuds dominants, arbres recouvrants, etc).

Dans la suite, nous présentons l'architecture d'un cadre d'interface pour implémenter des contraintes prêtes à brancher. Ensuite, nous illustrons une telle interface avec les contraintes `boundAllDiff` [LOQTvB03], qui est une contrainte globale simple (dans le sens qu'elle ne fait intervenir aucune structure de données interne), et `tree` [BFL05a], qui est une illustration typique d'une contrainte à état explicite (la structure de données interne représente un graphe orienté et des propriétés portant sur ce graphe). Finalement, nous branchons ces deux contraintes à deux solveurs différents.

7.4 Modèle des contraintes prêtes à brancher

Cette section étudie un modèle générique pour les contraintes prêtes à brancher. L'architecture est résumée par la figure 7.13. Elle est basée sur une décomposition en trois étapes, largement inspirée par le bien connu pattern *Observer* du génie logiciel orienté objet.

- Une *Interface de Solveur* propose un ensemble de méthodes pour traduire les événements des variables du solveur en événements génériques gérables par l'interface dans la contrainte. Plus précisément, le solveur traduit les événements survenant sur les variables en événements génériques, et soumet ces événements au gestionnaire. Par exemple, supposons que le domaine d'une variable a changé, des valeurs ont été retirées (durant le filtrage ou l'énumération) ou ont été ajoutées (à cause d'un retour-arrière), alors un nouvel événement générique est généré et

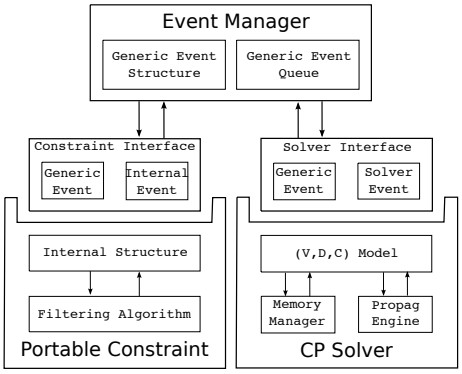

FIGURE 7.13: Interfacage d'un solveur et d'une contrainte *prête à brancher* par l'intermédiaire du gestionnaire d'événements.

transmis au gestionnaire d'événements.

- Une *Interface de Contrainte* propose un ensemble de méthodes pour traduire les événements génériques fournis par le gestionnaire en événements compréhensibles par les structures de données internes de la contrainte. Les décisions de retrait (sur la structure interne) effectuées par les algorithmes de filtrage sont traduites de manière symétrique en événements génériques et sont renvoyés au gestionnaire.
- Un *Gestionnaire d'événements* qui distribue chaque événement générique. Il peut être vu comme une structure observable qui notifie les nouveaux éléments postés dans la queue d'événements génériques de l'interface de la contrainte prête à brancher correspondante ou de l'interface du solveur sous-jacent.

Typiquement, si une variable du solveur a été instanciée, l'interface du solveur construit et poste un nouvel événement générique dans la queue du gestionnaire. Ensuite, le gestionnaire notifie l'interface de la contrainte que de nouveaux événements sont disponibles. De manière similaire, si l'algorithme de filtrage change la structure interne de la contrainte alors, un événement générique est produit et posté dans la queue du gestionnaire. L'interface du solveur est alors notifiée qu'un nouvel événement est disponible. Cependant, on peut remarquer que même si une traduction d'un événement du solveur en un événement contrainte est généralement direct, l'opération inverse peut être compliquée. Par exemple, dans le cadre des contraintes basées sur des graphes, les événements haut niveau du solveur peuvent être des retraits de valeurs dans les domaines des variables. De tels événements sont interprétés comme un ensemble de retraits d'arc dans la structure de données modélisant le graphe. De manière similaire, un retour-arrière consiste en un ajout d'un ensemble de valeurs dans les domaines des variables, et est interprété comme l'ajout d'un ensemble d'arcs dans la structure de données modélisant le graphe.

Dans la suite, nous ne détaillerons pas l'implémentation de notre interface générique mais nous détaillerons les différentes difficultés qui peuvent être rencontrées en fonction du type de contrainte et des algorithmes de résolution. Nos expérimentations sont basées sur les solveurs `Choco` et `Gecode`. Nous avons décidé volontairement de considérer deux types de contraintes globales : la contrainte `boundAllDiff` qui n'utilise aucune structure de données internes et la contrainte `tree` dont la structure de données interne est basée sur une représentation de graphe qui implique de véritables traductions des événements touchant les domaines en événements liés au graphe.

7.4.1 Une interface pour une contrainte globale simple : le cas `boundAllDiff`

Nous montrons d'abord, que dans le cas des contraintes globales qui ne requièrent pas de structures de données backtrackables (c.-à-d., toutes les inférences sur les domaines sont faites par un raisonnement direct sur les domaines des variables) alors, proposer une version prête à brancher de ces contraintes est facile. Basé sur l'algorithme de consistance de bornes introduit dans [LOQTvB03], cette section présente une implémentation prête à brancher de la contrainte `boundAllDiff`. Nous soulignons que la difficulté d'implémenter une telle contrainte prête à brancher est uniquement liée à l'interface avec le solveur.

Soit $v_1, v_2, \ldots, v_n$ un ensemble de variables avec les domaines finis $dom(v_1)$, $dom(v_2), \ldots,$ $dom(v_n)$, et des contraintes de différences deux à deux dont l'ensemble est noté $\texttt{allDiff}(v_1, \ldots, v_n)$ défini par $\{(e_1, \ldots, e_n) \mid \forall e_i \in dom(v_i), e_i \neq e_j \text{ for } i \neq j\}$. Il s'agit d'une contrainte très étudiée. Le premier algorithme de filtrage pour rendre les domaines consistants a été introduit par Régin dans [Rég94]. Ici, nous rappelons l'algorithme de filtrage propageant la consistance aux bornes introduit dans [LOQTvB03]. Afin de fournir des approches de filtrage rapides et pouvant s'appliquer à des problèmes de grande taille, plusieurs algorithmes de filtrage (qui atteignent un niveau de consistance plus faible que la consistance de domaine) ont été introduits. Le plus rapide est l'algorithme de consistance aux bornes, `boundAllDiff`. Intuitivement, au plus n variables peuvent avoir leur domaine inclus dans un intervalle contenant n valeurs. Dans un intervalle de Hall (un intervalle I de taille n, tel qu'il y ait n variables dont le domaine est contenu dans I), une solution utilisera toutes les valeurs pour ces variables rendant ses valeurs inutilisables pour les autres variables. Ainsi, pour maintenir la consistance aux bornes, nous devons vérifier pour chaque intervalle I qu'il y a, au moins, autant de valeurs que de variables dont le domaine est inclus dans I. Et pour chaque intervalle de Hall I, nous devons interdire toutes les valeurs de I dans les autres variables.

Définition 23 (`boundAllDiff` – consistance)**.** Une contrainte `boundAllDiff`, portant sur les variables $(x_1, \ldots, x_n)$, est consistante aux bornes si et seule-

ment si les conditions suivantes sont vérifiées : (1) $|D_i| \geq 1 (i = 1, \ldots, n)$, (2) pour chaque intervalle I : $|K_I| \leq |I|$, et (3) pour chaque intervalle de Hall I, $\{min(x_i), max(x_i)\} \cap I = \emptyset$ for all $x_i \notin K_I$.

Pour chaque réveil, la contrainte `boundAllDiff` trie les variables et initialise plusieurs compteurs, ainsi aucune structure de données n'est maintenue entre deux appels de la procédure de filtrage. Les informations requises par la contrainte est l'ensemble des variables avec les valeurs maximum et minimum de chaque domaine. Alors, pour être capable de brancher la contrainte `boundAllDiff` dans différents solveurs, l'interface doit fournir une vue générique des variables. Ainsi, pour une contrainte `boundAllDiff` prête à brancher, le cadre générique nécessite l'implémentation de :

1. L'interface du solveur donne le minimum et le maximum du domaine de chaque variable et, envoie une vue de ces domaines au gestionnaire d'événements. Symétriquement, les événements de la contrainte envoyés par le gestionnaire d'événements sont traduits en événements de solveur et sont finalement convertis en modification de domaine des variables.
2. Le gestionnaire d'événements peut être vu ici comme une fonction bijective qui convertit les événements solveur en événements contrainte et vice-versa.
3. L'interface contrainte contient l'algorithme de filtrage `boundAllDiff`. Les événements contrainte générés par l'algorithme de filtrage consistent uniquement en un ensemble de modifications portant sur les valeurs minimale et maximale des vues des variables dans la structure de la contrainte.

La complexité de l'interface dépend des informations requises par la contrainte et des informations disponibles dans l'algorithme de recherche. Généralement, les valeurs maximale et minimale pour une variable sont accessibles en temps constant, aussi l'interface a une complexité de $O(n)$. De plus, la structure de données et l'algorithme de filtrage sont exactement les mêmes dans la version prête à brancher ou dans la version *ad hoc* de la contrainte.

7.4.2 Le cas des contraintes globales à états : la contrainte `tree`

Dans le cas de contraintes globales qui reposent sur une structure de données backtrackable (c.-à-d., quelques inférence ont besoin de détecter de manière répétée un motif caché dans les domaines des variables) alors, proposer une version prête à brancher de ces contraintes est plus délicat, et des algorithmes *pleinement dynamiques* sont nécessaires pour fournir une contrainte indépendante du solveur. De plus, comme une contrainte prête à brancher n'est pas dépendante du solveur hôte, un choix plus large de structures de données est offert. Cette liberté permet de choisir et d'implémenter la plus efficace. Cette section illustre ce point avec la contrainte `tree` [BFL05a].

Le goulot d'étranglement de la complexité de la contrainte `tree` est lié au fait de maintenir de manière répétée plusieurs propriétés de graphe (composantes fortement connexes – cfc, fermeture transitive, nœud dominant) du graphe orienté $\mathcal{G}$ entre deux étapes de recherche. De manière basique, une nouvelle approche implémentant une telle contrainte et respectant la séparation des préoccupations introduite dans la figure 7.13, peut être décomposée de la manière suivante : premièrement, *l'interface de la contrainte* est décomposée en une structure de la contrainte et en un algorithme de filtrage. La *structure de la contrainte* est basée sur une structure de données générique, incrémentale en ajout et en retrait, qui modélise le graphe orienté $\mathcal{G}$ et ses propriétés associées (c.-à.-d., cfc et la fermeture transitive). Cette partie contient les primitives permettant de mettre à jour la structure de données en fonction des retraits et des ajouts d'arcs. Ces primitives, pour résumer, calculent chaque propriété en ne considérant qu'un graphe partiel nécessaire [1] du graphe d'origine $\mathcal{G}$. *L'algorithme de filtrage*, basé sur les propriétés maintenues par le graphe (représenté par la structure de la contrainte), retire les arcs de $\mathcal{G}$ qui sont inconsistants avec la contrainte `tree`. Deuxièmement, *l'interface du solveur* et le *gestionnaire d'événements* qui assurent une relation bidirectionnelle entre les événements survenant sur le domaine des variables (c.-à-d. retraits/-restaurations des valeurs dans les domaines) et des événements survenant dans le graphe orienté $\mathcal{G}$ (retrait/ajout d'arcs).

Dans l'implémentation actuelle de la contrainte `tree` prête à brancher, chaque fois qu'un événement survient dans le domaine d'une variable, cet événement est d'abord interprété par l'interface du solveur pour produire un événement interne. Ensuite, la structure de données interne est mise à jour avec cet événement interne. Alors, l'algorithme de filtrage dédié à la contrainte `tree` est appliqué et les événements internes résultants sont traduits en événements génériques et sont envoyés au gestionnaire d'événements. Nous pouvons maintenant parler du gestionnaire d'événements. À partir des domaines des variables impliquées dans un CSP, généralement, quatre types d'événements peuvent être distingués : le retrait de valeurs dans les domaines, l'instanciation de variables, la mise à jour d'une borne inférieure et la mise à jour d'une borne supérieure. Cependant, les instanciations et les mises à jour de bornes peuvent facilement être décomposées en un ensemble de retraits. Ainsi, pour chaque type d'événement envoyé par l'algorithme de recherche à l'interface du solveur, une traduction de l'événement en un ensemble de retraits dans la structure de la contrainte est réalisée. Cependant, tous les retraits d'arcs ne sont pas gérés de la même manière par le gestionnaire d'événements. En effet, l'événement à l'origine de l'ensemble des retraits est considéré afin d'améliorer l'efficacité des mises à jour de la structure de la contrainte. Par exemple, un ensemble de retraits liés à un événement d'instanciation surve-

1. Étant donné un graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, un *graphe partiel* $\mathcal{G}'$ de $\mathcal{G}$ est défini par $(\mathcal{V}' \subseteq \mathcal{V}, \{(i,j) \in \mathcal{E} \mid i,j \in \mathcal{V}'\})$

boundAllDiff		nReines					
		25	50	75	100	150	200
prête à brancher		22	35	1538	2461	5741	14217
ad-hoc		19	35	1531	2435	5711	14006

TABLE 7.2: Comparaison des temps de calcul (ms) pour les versions prête à brancher et *ad hoc* de la contrainte `boundAllDiff`.

nant sur une variable entraîne une modification locale dans le voisinage du nœud correspondant dans le graphe orienté $\mathcal{G}$ associé avec la contrainte. Cette information peut être prise en compte dans le but de réaliser ces modifications efficacement.

7.5 Évaluation

Dans cette partie, nous nous proposons d'évaluer le comportement des contraintes *prêtes à brancher*. Toutes les expérimentations ont été réalisées avec `choco` (version 1.2.04) et `gecode` (version 1.3.1) sur un processeur Intel Xeon cadencé à 2.4GHz et possédant 1Go de RAM, mais dont seulement 128Mo alloués à la machine virtuelle java.

Cette évaluation est réalisée en deux étapes. La première (section 7.5.1), évalue le surcoût engendré par le découplage. Pour cela, nous nous focalisons sur la contrainte `boundAllDiff` dans le contexte des n reines (tableau 7.2). Puis, la contrainte `tree` montre la portabilité effective de sa version *prête à brancher* pour les solveurs `Choco` et `Gecode` (tableau 7.3). La deuxième étape (section 7.5.2), évalue le comportement des contraintes *prêtes à brancher* dans un environnement hétérogène (on retrouve à la fois des contraintes *ad hoc* et des contraintes *prêtes à brancher* dans le modèle). Pour cela, nous nous intéressons au problème de chemin hamiltonien (tableau 7.4).

7.5.1 Sur-coût lié à la portabilité

Nous évaluons le surcoût d'interfaçage des contraintes `boundAllDiff` et `tree` avec deux solveurs de contraintes : `Choco` et `Gecode`. L'interface traduit une information liée au solveur en des informations génériques. L'implémentation de cette interface dépend du niveau d'information fourni par le solveur. `Choco`, centré événement, est capable de fournir aux contraintes les modifications ayant provoqué leur réveil. `Gecode`, quant à lui centré propagateur, ne peut fournir cette information. Pour la contrainte `tree` branchée sur `Choco`, l'interface présente une complexité temporelle directement liée au nombre de modifications de domaines réalisées tandis que pour `Gecode`, l'interface doit passer en revue le domaine de chaque variable pour identifier les modifications. Dans le cas de `boundAllDiff`, l'interface n'a uniquement be-

Ordre du graphe	temps `Choco` (ms)		
	Interface	Contrainte	Structure
25	5	10	27
50	5	57	229
75	11	190	863
100	18	440	2287
150	50	1466	9524
200	82	3214	27551

Ordre du graphe	temps `Gecode` (ms)		
	Interface	Contrainte	Structure
25	6	10	27
50	24	56	228
75	58	186	879
100	120	439	2310
150	368	1329	9596
200	812	3009	27097

TABLE 7.3: Temps de calcul pour `tree` branchée sur les deux solveurs (Choco et Gecode).

soin de connaître les valeurs maximale et minimale des domaines des variables aussi bien pour l'un que l'autre des solveurs.

Nous résolvons les n reines à 25, 50, 75, 100, 150 and 200 reines. Le problème est modélisé à l'aide de 3 contraintes `boundAllDiff` (en version prête à brancher d'une part et en version *ad hoc* d'autre part). Cette contrainte ne gère aucune structure de données interne, ainsi le surcoût pour la version prête à brancher est minimal. Pour 200 reines, le temps passé dans les trois modules d'interface (une interface par contrainte) ne représente que 1.5% du temps global de calcul (tableau 7.2).

On peut noter qu'une contrainte *prête à brancher* peut-être utilisée avec n'importe quel problème. Le tableau 7.3 montre les détails du temps de calcul utilisé par les différentes composantes de la contrainte `tree`, tant pour `Choco` que pour `Gecode`. Pour des graphes d'ordre {25, 50, 75, 100, 150, 200}, et de densité {0.05, 0.2, 0.4, 0.5, 0.6, 0.8, 0.95}, 50 instances d'un problème de partitionnement de graphe sont générées (soit au total 2100 graphes). Notons que les deux solveurs utilisent ici la même heuristique de choix de variable. La colonne "Interface" du tableau 7.3 montre parfaitement le sur-coût engendré par l'interface dans Gecode. On note aussi ce que le temps passé dans la contrainte elle-même est le même dans les deux solveurs : c'est ce que montrent les colonnes "Contrainte" et "Structure".

7.5.2 Comportement en pratique

Cette section résume les résultats expérimentaux des contraintes prêtes à brancher utilisées pour résoudre un problème de chemin hamiltonien. Cette évaluation montre l'efficacité des contraintes prêtes à brancher dans un contexte pratique. Pour cela, le problème de chemin hamiltonien est modélisé

Ordre du graphe		contrainte `tree` à brancher	
		ad hoc `boundAllDiff`	à brancher `boundAllDiff`
25		56	52
50		6224	6235
75		10311	10126
100		14956	14855
150		27271	27380
200		45099	45919
Ordre du graphe		contrainte `tree` ad hoc	
		ad hoc `boundAllDiff`	à brancher `boundAllDiff`
25		57	58
50		5529	5577
75		10858	10745
100		19201	19267
150		60683	60612
200		156055	155793

TABLE 7.4: Résolution du problème de chemin hamiltonien combinant contraintes ad hoc et prêtes à brancher (ms).

à l'aide des contraintes `boundAllDiff` et `tree`.

L'évaluation est réalisée en observant toutes les combinaisons possibles (ad hoc/prête à brancher) des contraintes `tree` et `boundAllDiff` avec le solveur choco. Pour chaque ordre de graphe dans {25, 50, 75, 100, 150, 200}, et pour des densités {0.10, 0.25, 0.40, 0.50, 0.65, 0.75, 0.90}, 50 instances sont générées (soit globalement 2100 graphes). Ici, un timeout de 5 minutes a été fixé et une heuristique aléatoire de choix de variable a été utilisée.

Les résultats présentés dans le tableau 7.4 montrent qu'utiliser des contraintes prêtes à brancher n'a pas d'impact sur l'efficacité, même en cas de combinaison avec des contraintes *ad hoc*. Même, on peut voir que la version prête à brancher de la contrainte `tree` est bien meilleure que la version *ad hoc*. Ceci est principalement dû à la nature complètement incrémentale de cette contrainte (aucune structure backtrackable n'est utilisée). Enfin, la contrainte `boundAllDiff` dans sa version prête à brancher est équivalente à sa version *ad hoc*.

7.6 Discussion

Cet article tente d'identifier les limites du découplage d'une contrainte globale du solveur sous-jacent. Nous avons montré dans les paragraphes précédents qu'il n'y a pas de frein majeur à un tel découplage. En fait, cette question revient à lever deux interrogations :

7.6.1 Est-ce que l'implémentation d'une contrainte prête à brancher est difficile ?

Les contraintes globales sont généralement implémentées à l'aide des structures dédiées et de l'API d'un solveur donné. Ces contraintes, le plus souvent, embarquent des structures de données persistantes utilisées pour maintenir leur propre niveau de consistance pendant la recherche de solution. En réalité, cette interférence du solveur avec l'architecture interne de la contrainte est plutôt une limitation pour l'efficacité et l'expressivité d'une implémentation. Par exemple, les nouveautés proposées par les langages ne sont pas, la plupart du temps, directement utilisables étant donné le temps de latence induit par le solveur lui-même (par exemple, les *generics* en Java).

Ainsi, dans le contexte des contraintes prêtes à brancher la question des structures de données internes n'en est plus une car toute latitude est laissée au développeur. Par contre, la vraie question est la manière de mettre à jour la représentation interne de la contrainte face aux événements fournis par le solveur. Une réponse satisfaisante doit prendre en compte la complexité du maintien des propriétés maintenues par la représentation interne. Intuitivement, trois cas peuvent être distingués : dans le premier cas, les propriétés sont calculées à chaque appel de la contrainte sans aucun effet mémoire (comme dans la section 7.4.1). Ainsi, après chaque modification les propriétés sont recalculées et la contrainte peut les exploiter ; dans le deuxième cas, les propriétés ne peuvent être recalculées de manière efficace mais il existe des algorithmes complètement incrémentaux qui permettent de prendre en compte les modifications (cas de la section 7.4.2). Dans le cas des propriétés de graphe, par exemple, de nombreux travaux existent pour prendre en compte dynamiquement des modifications de structures. Ainsi, [Rég08] propose un algorithme de maintien d'un arbre recouvrant de poids minimal devant des ajouts/retraits d'arêtes ; dans le dernier cas, il est nécessaire d'embarquer dans la contrainte des mécanismes explicites de retour arrière (selon diverses méthodes : trailing, copie, recalcul). Là, il est tout de même nécessaire d'être capable de savoir depuis la contrainte si un retour arrière a eu lieu ou non.

7.6.2 Est-ce que réaliser l'interface pour une contrainte prête à brancher est difficile ?

Les solveurs manipulent des événements sur les variables décrivant des modifications des domaines : retraits de valeur(s), mise à jour de borne(s), etc. Ces événements sont compris et traités par toutes les contraintes portant sur les variables du solveur. Mais, pour les contraintes prêtes à brancher, ces événements sont généralement vides de sens *per se*. C'est pourquoi l'interface doit les traiter pour transformer ces événements du solveur en événements sur les structures de données internes de la contrainte.

Dans la première approche de découplage, les événements du solveur

peuvent traduit en *quelque chose a changé*. Cette information permet de recalculer les propriétés à maintenir.

Dans la seconde approche, les événements du solveur sont transformés en des mises à jour de la structure de données interne traitées ensuite par les algorithmes incrémentaux fournis par la contrainte. Ces mises à jour dépendent du type de modifications que sont capables de traiter les algorithmes en question. Par exemple, dans la section 7.4.2, les événements du solveur ont été transformés en des retraits (ou ajouts) d'arêtes dans le graphe de référence.

Enfin dans la dernière approche, les événements du solveur sont transformés en des événements compris par la structure backtrackable générique utilisée.

La limite principale au découplage des contraintes globale n'est pas une question de génie logiciel mais plutôt une question d'interopérabilité des langages de programmation utilisés pour implémenter les contraintes. Par exemple, les contraintes de choco sont développées en Java, celles de Gecode en C++, celles de Comet en Comet et les contraintes de Sicstus Prolog en C. Évidemment des outils existent pour traiter ces problématiques. Par exemple, nous avons utilisé JNI (Java Native Interface) pour interfacer nos contraintes prêtes à brancher avec Gecode (à travers le module Gecode/J). JNI permet à du code Java tournant sur une machine virtuelle d'appeler et d'être appelé par d'autres applications écrites dans d'autres langages de programmation. Les coûts engendrés par une telle interface supplémentaire sont liés au type d'arguments passés entre les différent langages. Mais, en faisant attention, les applications Java peuvent appeler du code natif de manière relativement efficace. De plus, le sur-coût de JNI devient négligeable quand le code natif est fort consommateur de cpu.

Dans le cas de notre interface, les paramètres sont des messages d'événements (sur les variables) ainsi JNI est une solution envisageable. Par contre, pour une application codée en C, appeler une fonction Java est peu efficace car il est alors nécessaire d'utiliser des mécanismes réflexifs dans Java [WK00]. Il existe d'autres approches pour assurer l'interopérabilité des langages : communication par TCP/IP ou par IPC (inter-process communication) par exemple. Les applications Java en particulier peuvent utiliser les technologies objet distribuées comme l'API IDL [2].

L'incrémentalité permet d'adapter l'état des contraintes aux évolutions des modèles. Lors de la modification du modèle afin de prendre en compte les évolutions on peut avoir besoin de restaurer l'état des variables. Les explications permettent de savoir quelles sont les valeurs dont la suppression est remise en cause et qu'il faut donc réintroduire dans les domaines des variables. Nous donnons ici l'exemple de l'explication de la contrainte globale `tree`.

2. http ://java.sun.com/docs/books/jni/html/intro.html

7.7 Explication de la contrainte `tree`

Grâce à la monotonie et l'incrementalité nous avons vu comment gérer la modification du modèle lorsque celle ci ne remet pas en cause les raisonnements effectués. Les explications permettent de traiter les cas où les modifications qui impliquent une remise en cause du filtrage effectué durant la recherche précédente. Au travers du cas de la contrainte `tree` nous illustrons les difficultés que représente l'explication du filtrage d'une contrainte.

Une explication correspond à l'ensemble des événements et propriétés responsables de la modification du domaine des variables. Ainsi lorsqu'on explique une contrainte, on identifie l'ensemble des éléments responsable du filtrage d'une valeur. Le but est de pouvoir restaurer les valeurs supprimées lorsqu'un des éléments responsable du filtrage de la valeur est remis en cause.

7.7.1 Explications liées au filtrage de la variable `ntree`

On observe cinq causes de mise à jour dans la contrainte `tree` : $\overline{\texttt{ntree}}$, $\underline{\texttt{ntree}}$, $max(\texttt{ntree})$, $min(\texttt{ntree})$ et les noeuds dominants. Expliquer la contrainte `tree`, signifie expliquer les causes de chacune de ces mise a jour.

$\overline{\texttt{ntree}}$ correspond au nombre de racines potentiels, pour expliquer la borne supérieur de `ntree` il faut donc expliquer pourquoi les arcs (u, u) ont disparus.

$$\text{expl}(max(\texttt{ntree})) = \bigwedge_{u \in \mathcal{G}, (u,u) \in \mathcal{E}_{inf}} \text{expl}((u, u))$$

$\underline{\texttt{ntree}}$ correspond au nombre de composantes puits, pour expliquer la borne inférieure de `ntree` il faut donc expliquer, pour chaque composante puits, la disparition de l'ensemble des arcs sortants. Soit $\mathcal{P}$ l'ensemble des composantes puits de $\mathcal{G}$.

$$\text{expl}(min(\texttt{ntree})) = \bigwedge_{\mathcal{S} \in \mathcal{P}} (\text{expl}(\mathcal{S} \text{ est une SCC}) \bigwedge_{\substack{w \in \mathcal{S}, \\ x \notin \mathcal{S}, \\ (w,x) \in \mathcal{E}_{inf}}} \text{expl}((w, x)))$$

Montrons :

$$\text{expl}(min(\texttt{ntree})) = \bigwedge_{\mathcal{S} \in \mathcal{P}} (\bigwedge_{\substack{w \in \mathcal{S}, \\ x \notin \mathcal{S}, \\ (w,x) \in \mathcal{E}_{inf}}} \text{expl}((w, x)))$$

Démonstration. Partitionnons l'ensemble des arcs de $\mathcal{S}$ en 4 ensembles : l'ensemble des arcs $\mathcal{X} = \{(u,v)/u \notin \mathcal{S}, v \notin \mathcal{S}\}$, $\mathcal{I} = \{(u,v)/u \in \mathcal{S}, v \in \mathcal{S}\}$, $\mathcal{E} = \{(u,v)/u \notin \mathcal{S}, v \in \mathcal{S}\}$ et $\mathcal{R} = \{(u,v)/u \in \mathcal{S}, v \notin \mathcal{S}\}$. Prouvons que les arcs appartenant à $\mathcal{E}$ ne jouent aucun rôle pour l'explication. Par l'absurde supposons que l'ajout d'un arc e de $\mathcal{E}$ invalide la condition, c'est à dire l'ajout de e entraîne l'ajout d'au moins un nœud dans $\mathcal{S}$. Autrement dit, l'ajout e crée un cycle, donc il existe un arc faisable de $\mathcal{R}$. Contradiction. □

7.7.2 Explications à partir de la variable ntree

Lorsque la variable `ntree` doit atteindre la valeur $\overline{\texttt{ntree}}$ l'algorithme force, pour chacune des racines potentielles, l'arc boucle. Ainsi pour chaque racine potentielle, l'explication du retrait de l'ensemble de ses arcs sortants sont les causes de l'affectation de la variable `ntree` à la valeur $\overline{\texttt{ntree}}$. Un nœud est racine potentielle par construction, il n'y a donc aucune explication qui lui est lié.

Dans le cas où `ntree` doit atteindre $\underline{\texttt{ntree}}$, l'algorithme retire, pour chaque racine potentielle qui n'appartient pas à une composante fortement connexe puits, sa boucle sur lui même. Pour chaque racine potentielle a qui on retire son arc boucle, les explications du retrait sont les causes de l'affectation de la variable `ntree` à la valeur $\underline{\texttt{ntree}}$. Un nœud est une racine potentielle et n'appartienne pas à une composante fortement connexe par construction, il n'y a donc aucune explication qui lui est lié.

7.7.3 Explications liées au filtrage sur les noeuds dominants

Les arcs infaisables sont les arcs sortant (j,k), où j est un nœud dominant, tel qu'il n'existe pas un chemin de i à une racine potentielle utilisant l'arc (j,k). Avec i un nœud pour lequel j domine toutes les racines potentielles par rapport à i. Ainsi l'explication du retrait des arcs sortant (j,k) tient au fait que si ils sont instanciés le nœud i ne peut plus accéder à une racine potentielle. Autrement dit l'explication du retrait sont les causes qui ont fait que j domine toutes les racines potentielles par rapport à i. Il faut donc expliquer la disparition de l'ensemble des chemins, ne passant pas par j, reliant le nœud i à une racine potentielle. Cela peut être réalisé en cherchant itérativement des chemins allant du nœud i à une racine potentielle dans un graphe possédant l'ensemble des arcs (même les filtrés) et du quel on a retiré le nœud j. Chaque fois qu'on a découvert un nouveau chemin on ajoute l'explication de la disparition de l'un de ses arcs (puisque ce chemin n'existe plus dans le graphe courant) à l'explication globale. Les arcs dont le retrait a été pris en compte sont retirés du graphe. On recommence jusqu'à ce que plus aucun chemin ne soit possible.

7.8 Explications sémantiques

On observe, dans le cas des explications liées au filtrage sur les noeuds dominants, que le calcul de l'explication peut être très couteux. De plus ce calcul est réalisé avant même de savoir si l'explication sera utile ou non. Dans le cadre dynamique les explications sont utilisées comme trace des déductions faites durant la propagation. Une expression aussi "basse" des explications n'est donc pas forcement utile. Notre idée est donc de rester à un niveau sémantique pour proposer des explications moins couteuses et plus efficaces.

La difficulté engendrée par le calcul des explications des noeuds dominants tien au fait, qu'on tente systématiquement de rapporter l'explication à des événements basiques (retrait de valeurs) du solveur. Il possible d'expliquer une déduction ou un retrait de valeur en restant au niveau des propriétés responsables des actions du solveur. Ainsi dans le cas des explications liées au filtrage sur les noeuds dominants une explication sémantique possible est : $expl(\neg(j,k)) = j$ domine i ou encore $expl(\neg(j,k)) = \bigwedge_{S \in P} expl(\neg S)$ avec P l'ensemble des chemins allant de i à une racine potentielle sans passer par le nœud j.

Le désavantage est que cela alourdit les calculs lors de la restauration d'une valeur, préalablement supprimée. Puisque chaque ajout de valeur doit être traduit en "conséquence pour les propriétés". Ainsi la restauration d'une valeur peut avoir pour conséquence qu'un nœud j ne soit plus dominant par rapport à un nœud i, le solveur doit alors restaurer toutes les valeurs dont le retrait s'explique par l'intervention de la propriété "j est dominant par rapport à i".

Ainsi les explications ne sont calculées que lorsque le solveur a effectivement besoin d'elles. L'idéal est de pouvoir maintenir l'ensemble des propriétés de manières incrémentales afin de limiter le surcout lié à leur calcul.

7.9 Conclusion

La monotonie permet l'ajout de variable aux contraintes et l'incrémentalité permet la modification de l'état des contraintes sans remettre complètement en cause les structures interne de la contrainte. L'utilisation de structures internes incrémentales a pour effet de rendre les contraintes indépendantes des structures proposées par le solveur. En partant de cette constatation on présente une méthode pour implémenter des contraintes indépendantes des solveurs et réutilisables. L'incrémentalité permet de maintenir l'état de la contrainte. Pour maintenir l'état du solveur et restaurer l'état des variables lors d'une perturbation qui remet en cause le filtrage effectué nous devons utiliser des explications. On illustre les difficultés liées à l'explication d'une contrainte au travers de la contrainte `tree`.

Troisième partie

Conclusion

Chapitre 8

Conclusions et perspectives

Dans cette thèse nous nous sommes penchés sur la question des problèmes dynamiques dans le cadre des problèmes de satisfaction de contraintes. Des solutions existent mais la prise en compte d'événements totalement imprévisibles (que ce soit par leur type ou par le moment où ils surviennent) et pouvant remettre en cause le modèle utilisé (suppression de contraintes, ajout de variable) sont encore peu traités. En proposant plusieurs outils notre but est de permettre de prendre en compte de manière efficace ces cas.

Ainsi après s'être posé la question de savoir ce qu'était une solution stable pour un utilisateur et avoir introduit plusieurs mesures pour prendre en compte son point de vue, nous nous sommes concentrés sur trois approches distinctes et complémentaires. La première approche repose sur l'utilisation d'une méthode de recherche locale pour réparer les solutions après perturbations. L'utilisation des explications dans ce contexte nous permet de proposer une nouvelle manière de définir le voisinage d'une solution. La deuxième approche, utilise les nogoods pour enregistrer les zones de l'espace de recherche dans les quelles il n'y a pas de solutions. Dans ce contexte nous proposons une contrainte, basée sur l'utilisation d'automates, permettant de stocker les nogoods de manière efficace et dynamique (l'ajout et le retrait d'un tuple sont incrémentaux). Grâce à plusieurs expérimentations nous avons mis en évidence l'efficacité de notre contrainte pour le filtrage. Enfin, la dernière approche utilise les propriétés des contraintes pour limiter le travail redondant lors de la perturbation du modèle. Après avoir prouvé l'intérêt de l'incrémentalité pour les contraintes globales en terme d'efficacité nous avons généralisé afin de proposer une méthode permettant de découpler une contrainte d'un solveur. Ces contraintes portables complètement incrémentales permettent aux réseaux de contraintes de s'adapter aux perturbations sans remettre en cause l'instanciation courante du problème.

A court terme, plusieurs autres expérimentations pourraient être menées pour confirmer l'efficacité de nos outils et pour les améliorer. Concernant, la contrainte automate le problème de trouver un bon ordre des variables pour permettre un stockage compact des tuples n'a pas encore de réponse. De même, dans le cas des contraintes portables de nombreuses question de génie logiciel se posent encore, en particulier pour permettre la communication efficace entre des éléments codés dans des langages de programmation différents.

Plus globalement, se pose la question de l'aspect incrémental des contraintes. En effet l'incrémentalité apporte plusieurs avantages. Premièrement, elle permet une exploration non arborescente de l'espace de recherche puisque les contraintes ne reposent plus sur la backtrack chronologique. Deuxièmement, elle rend les contraintes indépendantes des solveurs. Ainsi le travail effectué lors de l'implémentation d'une contrainte peut être réutilise. Troisièmement, l'incrémentalité permet de limiter le travail lors des mises à jour. Ainsi les contraintes reposant sur des structures incrémentalités peuvent se montrer plus efficaces. Mais la généralisation de l'incrémentalité et de la portabilité pose deux questions. En effet, l'utilisation de contraintes

portable ouvre la discussion sur le type d'événements géré par les solveurs : Doivent ils être bas niveau ? (événements sur les domaines des variables) ou doivent ils intégrer de la sémantique ? (événements sur le modèle). Afin de pouvoir prendre en compte le plus grand nombre possible de cas, les solveurs se basent sur des informations bas niveau, réintégrer des informations plus précises pourrait permettre de traitement plus fin de la propagation ou de l'utilisation des règles de filtrages (conditions sur le débranchement du filtrage d'une certaine contrainte etc).

Bibliographie

[ABvB01] Jonathan Sillito Adam Beacham, Xinguang Chen and Peter van Beek. Constraint programming lessons learned from crossword puzzles. In *In* Proceedings of the 14th Canadian Conference on Artificial Intelligence, pages 78–87, 2001.

[AFM02] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic csps application to configuration. *Artif. Intell.*, 135(1-2) :199–234, 2002.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows : Theory, Algorithms, and Applications*. Prentice Hall, February 1993.

[Bar01] Roman Barták. Dynamic global constraints : A first view. *CoRR*, 2001.

[Bar03] Roman Barták. Dynamic global constraints in backtracking based environments. *Annals of Operations Research*, 118 :101–119, 2003.

[Bes91] Christian Bessière. Arc consistency in dynamic constraint satisfaction problems. In *Proceedings of AAAI'91*, pages 221–226, 1991.

[Bes92] Christian Bessière. Arc-consistency for non-binary dynamic csps. In *Proceedings of ECAI'92*, pages 23–27, 1992.

[BFL05a] N. Beldiceanu, P. Flener, and X. Lorca. The *tree* constraint. In *CP-AI-OR'05*, volume 3524 of *LNCS*, pages 64–78, 2005.

[BFL05b] N. Beldiceanu, Pierre Flener, and Xavier Lorca. The tree constraint. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'05)*, Lecture Notes in Computer Science, pages 64–78, may 2005.

[BH06] Russell Bent and Pascal Van Hentenryck. A two-stage hybrid algorithm for pickup and delivery vehicle routing problems with time windows. *Comput. Oper. Res.*, 33(4) :875–893, 2006.

[BHZ05] Lucas Bordeaux, Youssef Hamadi, and L. Zhang. Propositional satisfiability and constraint programming : A comparative survey. Technical Report MSR-TR-2005-124, 2005.

[BM96] R. J. Bayardo and D. P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *National Conference on Artificial Intelligence (AAAI-1996)*, pages 298–304, 1996.

[BR97] C. Bessière and J.-C Régin. Arc consistency for general constraint networks : Preliminary results. In *In* Proceedings of IJCAI'97, pages 398–404, 1997.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8) :677–691, 1986.

[CdGL+99] Bertrand Cabon, Simon de Givry, Lionel Lobjois, Thomas Schiex, and Joost P. Warners. Radio link frequency assignment. *Constraints*, 4(1) :79–89, 1999.

[CJ06] Hadrien Cambazard and Narendra Jussien. Identifying and exploiting problem structures using explanation-based constraint programming. *Constraints*, 11 :295–313, 2006.

[DD88] Rina Dechter and Avi Dechter. Belief maintenance in dynamic constraint networks. In American Association for Artificial Intelligence, editor, *Proceedings of AAAI'88*, pages 37–42, 1988.

[Deb96] Romuald Debruyne. Arc-consistency in dynamic csps is no more prohibitive. In *In 8 th Conference on Tools with Artificial Intelligence (TAI'96)*, pages 299–306, 1996.

[Dec90] Rina Dechter. Enhancement schemes for constraint processing : Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41 :273–312, 1990.

[DFJ+03] Romuald Debruyne, Gérard Ferrand, Narendra Jussien, Willy Lesaint, Samir Ouis, and Alexandre Tessier. Correctness of constraint retraction algorithms. In *FLAIRS'03 : Sixteenth international Florida Artificial Intelligence Research Society conference*, pages 172–176, St. Augustine, Florida, USA, May 2003. AAAI press.

[EGI97] D. Eppstein, Z. Galil, and G. Italiano. *Dynamic graph algorithms*. CRC Press, 1997.

[FB00a] David W. Fowler and Kenneth N. Brown. Branching constraint satisfaction problems. Technical report, Dept. of Computing Science, Univ of Aberdeen, 2000.

[FB00b] David W. Fowler and Kenneth N. Brown. Branching constraint satisfaction problems for solutions robust under likely changes. In *Principles and Practice of Constraint Programming*, pages 500 – 504, 2000.

[FBMB90] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Commun. ACM*, 33 :54–63, 1990.

[FLS96] Hélène Fargier, Jérôme Lang, and Thomas Schiex. Mixed constraint satisfaction : A framework for decision problems under incomplete knowledge. In *Proceedings of the 13th National Conf. on Artificial Intelligence (AAAI'96)*, pages 175 – 180, 1996.

[FMG05] Boi Faltings and Santiago Macho-Gonzalez. Open constraint programming. *Artif. Intell.*, 161(1-2) :181–208, 2005.

[Gin93] Matthew Ginsberg. Dynamic backtracking. 1 :25–46, 1993.

[GM94] Matthew L. Ginsberg and David A. McAllester. Gsat and dynamic backtracking. *Journal of Artificial Intelligence Research*, 1 :25–46, 1994.

[GW06] Diarmuid Grimes and Richard J. Wallace. R.j. : Learning from failure in constraint satisfaction search. In *Learning for Search : Papers from the 2006 AAAI Workshop*, pages 24–31, 2006.

[HG95] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. pages 607–613. Morgan Kaufmann, 1995.

[HP91] Pascal Van Hentenryck and Thierry Le Provost. Incremental search in constraint logic programming. In *Special Issue on selected papers from ICLP-90*, pages 257–276, 1991.

[JB97] Narendra Jussien and Patrice Boizumault. Dynamic backtracking with constraint propagation – application to static and dynamic csps. In *CP97 Workshop on The Theory and Practice of Dynamic Constraint Satisfaction*, Schloss Hagenberg, Austria, 1 November 1997.

[KB03] G. Katsirelos and F. Bacchus. Unrestricted nogood recording in csp search. In *Proceedings CP 2003*, pages 873–877, 2003.

[KB05] G. Katsirelos and F. Bacchus. Generalized nogoods in csps. In *National Conference on Artificial Intelligence (AAAI-2005)*, pages 390–396, 2005.

[KPS98a] Philip Kilby, Patrick Prosser, and Paul Shaw. Implementation of lns for constrained vrps. APES report, 1998.

[KPS+98b] Philip Kilby, Patrick Prosser, Paul Shaw, Christopher Beck, Andrew Davenport, and Claude Le Pape. A comparison of traditional and constraint-based heuristic methods on vehicle routing problems with side constraints. *Constraints*, 5 :389–414, 1998.

[LK73] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2) :498–516, 1973.

[LMM+99] E. Lamma, Paola Mello, M. Milano, Rita Cucchiara, G. Gavanelli, and Massimo Piccardi. Constraint propagation and value acquisition : why we should do it interactively. *Proceedings of the Sixteenth International Joined Conference on Artificial Intelligence (IJCAI99)*, pages 468–477, 1999.

[LMMC97] E. Lamma, Paola Mello, M. Milano, and Rita Cucchiara. Interactive constraint satisfaction. Technical report, University of Bologna, 1997.

[LMS02] I. Lynce and J. Marques-Silva. The effect of nogood recording in MAC-CBJ SAT algorithms. Technical Report RT/04/2002., 2002.

[LOQTvB03] Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter van Beek. A Fast and Simple Algorithm for Bounds Consistency of the AllDifferent Constraint. In *IJCAI*, pages 245–250, 2003.

[LS07] Mikael Z. Lagerkvist and Christian Schulte. Advisors for incremental propagation. In *CP'07*, LNCS, pages 409–422, 2007.

[MF90] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of AAAI'90*, pages 25–32, Boston, MA, 1990. AAAI Press.

[MJPL90] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of AAAI'90*, pages 17–24, Boston, MA, 1990. AAAI Press.

[MJPL92] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts : a heuristic repair method for constraint satisfaction and scheduling problems. *Artif. Intell.*, 58(1-3) :161–205, 1992.

[MMZ+01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[NB94] B. Neveu and P. Berl. Maintaining arc consistency through constraint retraction. In *Proceedings TAI94, IEEE*, pages 426–431. Press, 1994.

[NMST07] Beldiceanu Nicolas, Carlsson Mats, Demassey Sophie, and Petit Thierry. Global constraint catalogue : Pastn present and future. *Constraints*, 12 :21–62, 2007.

[Pes04] G. Pesant. A regular language membership constraint for finite sequences of variables. In LNCS Springer, editor, *Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258, 2004.

[Pro93] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9, 1993.

[Pro95] P. Prosser. MAC-CBJ : maintaining arc consistency with conflict-directed backjumping. Technical Report Research Report/95/177, Dept. of Computer Science, University of Strathclyde, 1995.

[PSF04] Laurent Perron, Paul Shaw, and Vincent Furnon. Propagation guided large neighborhood search. In Mark Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 468–481. Springer Berlin / Heidelberg, 2004.

[RÓ4] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI '94 : Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, volume vol. 1, pages 362–367. American Association for Artificial Intelligence, 1994.

[RD01] Christian Bessière Romuald Debruyne. Domain filtering consistencies. *Journal of Artificial Intelligence Research (JAIR)*, 2001.

[Ref04] Philippe Refalo. Impact-based search strategies for constraint programming. In *Proceedings of Principles and Practice of Constraint Programming - CP 2004*, 2004.

[Rég94] J.-C. Régin. A filtering algorithm for constraints of difference in CSP. In *AAAI'94*, pages 362–367, 1994.

[Rég08] Jean-Charles Régin. Simpler and incremental consistency checking and arc consistency filtering algorithms for the weighted spanning tree constraint. In *CPAIOR'08*, LNCS, pages 233–247, 2008.

[Sch94] Thomas Schiex. Solution reuse in dynamic constraint satisfaction problems. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 307–312. AAAI Press, 1994.

[Sch99] Christian Schulte. Comparing Trailing and Copying for Constraint Programming. In *ICLP'99*, pages 275–289, 1999.

[SD91] Eddie Schwalb and Rina Dechter. Temporal constraint networks. *Artificial Intelligence*, 49 :61–95, 1991.

[SF98] Mihaela Sabin and Eugene C. Freuder. Detecting and resolving inconsistency and redundancy in conditional constraint satisfaction problems. In *Web-published papers of the CP'98 Workshop on Constraint Problem Reformulation*. AAAI Press, 1998.

[SGL97] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Incremental Computation of Dominator Trees. *ACM Transactions on Programming Languages and Systems*, 19(2) :239–252, March 1997.

[Sha98] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Principles and Practice of Constraint Programming*, pages 417–431. Springer-Verlag, 1998.

[SV94] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problem. *International Journal of Artificial Intelligence Tools*, 3(2) :187–207, 1994.

[SV95] Thomas Schiex and Gerard Verfaillie. Maintien de solution dans les problèmes dynamiques de satisfaction de contraintes : bilan de quelques approches. In *Revue d'intelligence artificielle*, volume 9, pages 269–309, 1995.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2) :146–160, 1972.

[vHR06] Willem Jan van Hoeve and Jean-Charles Régin. Open constraints in a closed world. In *CPAIOR*, pages 244 – 257, 2006.

[VJ05] Gerard Verfaillie and Narendra Jussien. Constraint solving in uncertain and dynamic environments : A survey. *Constraints*, 10(3) :253–281, July 2005.

[Wal02] Toby Walsh. Stochastic constraint programming. In *Proceedings of the 15th ECAI, European Conference on Artificial Intelligence. IOS*, pages 111–115. Press, 2002.

[WK00] Steve Wilson and Jeff Kesselman. *Java Platorm Performance : Strategies and Tactics*. Addison-Wesley, 2000.

[WN99] Laurence A. Wolsey and George L. Nemhauser. *Integer and Combinatorial Optimization*. Wiley-Interscience, November 1999.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

Outillage logiciel pour les problèmes dynamiques

Guillaume Richaud

Résumé : En août 2005, British Airways mit quatre jours à rétablir ses vols après une grève d'une journée d'un de ses sous-traitants. En interconnectant et intégrant leurs systèmes d'aide à la décision, les entreprises deviennent de plus en plus soumises aux changements. Parallèlement, avec le développement de technologies comme les puces RFID et la localisation par GPS les entreprises sont capables de suivre en temps réel le déroulement des opérations sur le terrain.

Dans cette thèse nous nous intéressons au cas des problèmes de satisfaction de contraintes dans un cadre dynamique. En effet, depuis de nombreuses années la programmation par contraintes a fait la preuve de son efficacité pour résoudre des problèmes d'optimisation (tournées de véhicules, ordonnancements, etc) dans le cadre statique. Cependant le contexte dynamique soulève encore de nombreuses difficultés. Nous proposons donc un ensemble d'outils permettant la gestion et la prise en compte des événements, survenant de manière inattendue, dans le cadre de la programmation par contraintes. Chaque outil repose sur une approche particulière des problèmes dynamiques (tuples interdits, recherche locale, explications) et offre ainsi un éclairage différent et complémentaire.

Mots-clés : programmation par contraintes, problèmes dynamiques, stabilité, robustesse, explications, nogood

Abstract : In August 2005, British Airways took four days to resume its operations after a one day strike of one of its subcontractors. Because of the interconnection and the integration of their decision support, companies become more and more sensitive to modifications. At the same time, with new technologies such as RFID or GPS, companies are able to monitor operations in real time.

In this thesis, we consider constraint satisfaction problems in a dynamic context. Indeed, for many years constraint programming has proven its efficiency to solve optimization problems (vehicle routing, scheduling, ...) in static contexts. However, dynamic contexts raise many difficulties. We put forward a set of tools to manage and to take unexpected events into account in a constraint programming framework. Each tool is based on a specific approach for dynamic problems (forbidden tuples, local search, explanations) and gives a different and complementary overview of dynamicity.

Keywords : constraint programming, dynamic problems, stability, robustness, explanations, nogood